Arrows for Knowledge-Based Circuits

Peter Gammie

A thesis submitted for the degree of Doctor of Philosophy
of The Australian National University.

February 2013

Australian National University

NATURAM PRIMUM COGNOSCERE RERUM

College of Engineering and Computer Science

Declaration

The work in this thesis is my own except where otherwise stated.

Peter Gammie

Acknowledgements

I thank my parents, Liz and Richard, for their imperturbable support, and Clem Baker-Finch and John Lloyd for their encouragement and perspective. I am grateful to Kai Engelhardt for patiently rubbishing all sorts of half baked stuff over the years, and also to Pete Rickwood, Andrew Taylor and David Wahlstedt for many valuable discussions.

I am indebted to Gerwin Klein for teaching me what I always wanted to know about a proof assistant, and have benefited from the expertise of Florian Haftmann, Brian Huffman, Andreas Lochbihler and others on the Isabelle mailing list.

I have greatly enjoyed discussing synchronous programming languages with Tim Bourke and Grégoire Hamon over many years.

The debt this project owes to Ron van der Meyden is obvious. He introduced me to formally reasoning about knowledge while we worked on the MCK system; I build on that experience here. John Hughes taught me about Arrows at the Advanced Functional Programming Summer School in Estonia in 2004. I thank Antti Valmari for his expert advice on automata minimisation.

I value the insightful comments of Andreas Abel, Tim Bourke, Raj Goré, David Greenaway, Oleg Kiselyov, Ben Lippmeier, Adam Megacz, Toby Murray, Bernie Pope, Colin Runciman and Josef Svenningsson on various parts of this work.

Some of this work was carried out while I was an Australian Youth Ambassador for Development in T.P. Hồ Chí Minh, Việt Nam, funded by the Australian Government via AusAID. I thank Bạch Việt, Bích, Cử, Daz, Dung, Mai, Mike, Nhu, anh Thien, Tigôn and especially Loan for a year of welcome distractions. Cảm ơn nhiều! I am also grateful to June Andronick, Gerwin Klein, Toby Murray, Thomas Sewell and Simon Winwood of the L4.verified team at NICTA[1] for a pleasant six months of proofs and tea in the afternoon. I thank Andrew Taylor for sending paid work my way, and the Australian National University for the funding to complete this project.

> Our last conclusion is to recall a principle that has been so often fruitful in Computer Science and that is central to Scott's theory of computation: a good concept is one that is closed
>
> 1. under arbitrary composition
> 2. under recursion.
>
> — Gilles Kahn (1974)

[1] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

Abstract

Knowledge-based programs (KBPs) are a formalism for directly relating agents' knowledge and behaviour in a way that has proven useful for specifying distributed systems. Here we present a scheme for compiling KBPs to executable automata in finite environments with a proof of correctness in Isabelle/HOL. We use Arrows, a functional programming abstraction, to structure a prototype domain-specific synchronous language embedded in Haskell. By adapting our compilation scheme to use symbolic representations we can apply it to several examples of reasonable size.

Contents

Chapter 1

Introduction

THAT reasoning about distributed systems can be quite subtle is exemplified by the extensive literature on this topic, and the myriad models that have been proposed. Some of this complexity is due to the various constituents of such a system being able to observe only some of its global state. This often leads designers of such systems to informally ascribe knowledge of global properties to individual processes, such as in these excerpts from a classic text on the subject:

> ... The algorithm is based on what processes know about each other's initial values and on what they know about each other's knowledge of the initial values, and so on. ...
>
> — Lynch (1996, §5.2.2)

> ... To obtain the strongest result, we want to allow the adversary to be as powerful as possible; thus we assume that, when making its decisions about who takes the next step and when, it has *complete knowledge* of the past execution, including information about process states and past random choices. ...
>
> — Lynch (1996, §11.4.2)

These examples suggest that a formal treatment of components' knowledge might simplify the design task; concretely, we might specify our distributed system using explicit appeals to the knowledge of its constituents, and then derive an executable implementation. This approach has been explored by Fagin, Halpern, Moses, and Vardi (1997); Halpern and Zuck (1992) amongst many others, and here we develop a tool that mechanises parts of this process.

Philosophers have been discussing formal analyses of knowledge for millennia; for instance Socrates gave a series of definitions in a dialogue recounted by Plato (1987). This subject of *epistemology* remains central to modern philosophy, with the application of mathematical techniques to ancient problems being a major contribution of the twentieth-century analytic tradition. In particular, modern accounts of *modal logic* based on Kripke semantics model how agents might reason about mutual states of knowledge (Chellas 1980; Hintikka 1962; Lenzen 1978). An analysis of *social conventions* gave rise to a formal notion of *common knowledge* (Lewis 1969), which was linked to bargaining problems in economics (Aumann 1976) and later, the coordination problems that lie at the heart of distributed systems (Fagin, Halpern, Moses, and Vardi 1995). The underlying semantics has been used to demonstrate the unsolvability of some of these problems by showing that processes cannot acquire enough information to behave correctly (Lynch 1996). It has also been used to justify lower bounds on the amount of communication required to solve other problems (Fagin et al. 1995, Chapter 6).

The suggestion that distributed systems be designed by explicitly relating their actions to their epistemic state also links this engineering task with a venerable artificial intelligence trope: what an agent does should depend on its knowledge, and what an agent knows depends on what it does (Shoham and Leyton-Brown 2008). Typically agents are also imbued with *intentions* or *goals*, which we instead ascribe to the system designer and do not reason about explicitly.

We follow Fagin et al. (1997) in linking knowledge and action by embedding explicitly epistemic tests in otherwise standard programs. This increase in expressiveness comes at the cost of direct executability, and as we will see it may take significant effort to find corresponding standard programs that have the required behavior. Previous techniques for finding such implementations of these *knowledge-based programs* (KBPs) are manual, and mostly involve pencil-and-paper derivations of small systems (Bickford, Constable, Halpern, and Petride 2009; Engelhardt, van der Meyden, and Moses 2001; Halpern and Zuck 1992). Here we build on the work of van der Meyden (1996b) by algorithmically constructing implementations of KBPs in some finite-state scenarios. This fully-automatic technique handles only a limited class of systems but still admits some interesting examples. Justifying this algorithm is the topic of the first part of this thesis.

As our goal here is to show that knowledge is a useful abstraction for system design in practice, the second part of the thesis is concerned with an efficient implementation of this theory. The algorithm itself can be recast in terms of symbolic representations, specifically the *Boolean decision diagrams* (BDDs) familiar from temporal *model checking* (Clarke, Grumberg, and Peled 1999), which can greatly reduce space requirements. We also need a notation that allows scenarios to be flexibly described; often these are parametrised, and as we later show, adjusting the staging of the model construction can greatly reduce the running time of the algorithm. Moreover it is advantageous to have a prototype that can be easily and modularly extended.

Given that we are working in a finite-state setting, we adapt techniques for describing digital circuits to our task of describing KBPs and their environments. Specifically we take the time-honoured approach of rendering these using functional programming techniques, which avails us of a very expressive meta-language (Elliott, Finne, and de Moor 2003). We further justify our choice of implementation platform later in this chapter.

In summary, this thesis explores the use of knowledge as a formal construct in the design and implementation of finite-state systems expressed as circuits in the pure non-strict functional programming language Haskell (Peyton Jones 2003). We build on two traditions: formal reasoning about knowledge using modal logic, and describing circuits as functional programs.

We now briefly review these two traditions in more depth, and then provide a synopsis of the rest of this document.

1.1 Knowledge in system design

The idea of writing programs and specifications containing explicit tests for knowledge was ambient in the distributed systems community and literature of the 1980s; see Fagin et al. (1995) for a book-length account. The goal was to lift the abstraction level at which protocols are specified, with the expectation that this would help eliminate the details in correctness proofs that obscure the essential arguments. Halpern and Zuck (1992) used this approach to give uniform proofs of correctness for several classical bit transmission protocols such as the alternating bit protocol and Stenning's protocol. Baukus and van der Meyden (2004) gave an account of cache coherence which we discuss at more length in §6.6. Fagin et al. (1995, Chapter 7) present several examples in this style, including the Robot example that we describe at the start of Chapter 2.

It has long been realised that the design and analysis of *asynchronous* algorithms is much more difficult than for *synchronous* ones. (A synchronous system is one that proceeds in rounds, i.e., where there is a global clock that regulates communication between the distinct components of the system. In contrast components can evolve independently in asynchronous settings.) Therefore it has been suggested that we begin by designing a solution to our problem in a synchronous setting and then look for a way to map that solution to the implementation architecture. If the latter is asynchronous we might use a *synchroniser* as advocated by Awerbuch (1985). Lynch (1996) provides a book-length treatment of this methodology.

A partial reason for this difference in complexity between these two accounts of time is that the mutual states of knowledge amongst the components of a system is much simpler in synchronous settings (van der Meyden 1998). Indeed, synchronisation can make the difference between a solvable and an unsolvable problem, with the classical example being achieving consensus amongst a set of processes (see Fagin et al. (1995, Chapter 6) and Lynch (1996, Chapter 12)).

Here we work only with synchronous scenarios, and moreover restrict these to be finite-state. The latter allows us to use state traversal techniques when constructing implementations of our knowledge-based programs (KBPs), and it is particularly reasonable to ask that these implementations also be finite-state if we expect them to continuously react to its environment, as we discuss further in the next section.

In constraining the scenarios we treat, we trade generality for automation. For instance, instead of verifying that our (specification) KBPs have the desired properties as one does when using refinement techniques (Engelhardt et al. 2001; Halpern and Zuck 1992), we verify that the implementations constructed by the algorithms are satisfactory by model checking (Clarke et al. 1999). We hope this machine assistance will encourage more exploration of the KBP formalism.

The theory of KBPs has been extensively investigated (Fagin et al. 1997; van der Meyden 1996b), but little has been developed in the way of tool support for finding implementations; the recent work of Bickford et al. (2009) partially mechanises the process in a proof assistant, but is not automatic. In contrast van der Meyden (1996a,b,c) has shown some decidability results for these problems, and also that they are computationally difficult and cannot be uniformly solved. Within our synchronous setting there are some classes of KBPs for which implementations can be automatically constructed. These classes encompass many systems, including many board and card games as well as the *reactive systems* and *synchronous digital circuits* that we discuss further in the next section.

We present more background on formal reasoning about knowledge and KBPs in Chapter 2.

1.2 Reactive systems and synchronous digital circuits

The considerations of the previous section drive us to consider techniques for designing and analysing synchronous systems. These have been studied at length in several settings.

Firstly, the class of *reactive systems* was identified during the 1980s as a particularly tractable one (Berry 1989; Manna and Pnueli 1992). Berry (1999b) defines these as follows:

> "Reactive systems", also called "reflex systems", continuously react to stimuli coming from their environment by sending back other stimuli. Contrarily to interactive systems[1], reactive systems are purely input-driven and they must react at a pace that is dictated by the environment. Process controllers or signal processors are typical reactive systems.

For these systems a string of *domain-specific programming languages* (DSLs) embodying the *synchrony hypothesis* (Benveniste, Caspi, Edwards, Halbwachs, Le Guernic, and de Simone 2003) have proven very effective as they allow logical and real-time behaviour to be treated separately. We can verify that particular systems have particular properties using powerful and highly automated model checking tools (Clarke et al. 1999) which are based on various *temporal logics*. From this perspective our goal is to increase the expressiveness of synchronous languages by adding knowledge conditionals, and as a side effect, to provide a way to analyse information flow in such systems.

[1] An *interactive system* is one that takes control of the interaction. Berry cites operating systems, databases and the internet as examples.

Secondly, a natural example of a reactive system is a synchronous digital circuit, which is expected to respond to each tick of a clock by producing new inputs to the state-holding elements and outputs to the rest of the system as a function of their previous state and inputs. Clocks were introduced to abstract from signal propagation delays, which allows these types of circuits to be designed compositionally. This type of hardware is clearly finite-state, and one might argue that any implementation of a reactive system must satisfy the same restriction; if not, the system will eventually run out of space, or assuming a reasonable timing model for storage, take too long to respond.

Given our desire for a flexible substrate for our implementation, we develop our prototype in the long tradition of representing such circuits in lazy functional programming languages (Bird and Wadler 1988; Hughes 1989; Peyton Jones 2003) which we review at length in Chapter 4. The idea of *embedding* a DSL into an expressive meta-language is an old one (Hudak 1996; Landin 1966) and is increasingly useful as programming language technology matures. We note that most existing work has focussed on describing *data-oriented* circuits, where regularity is elegantly captured by higher-order functions, in contrast to the *control-oriented* circuits that we use to represent KBPs. As we discuss in Chapter 4, it pays to be cautious about the semantic foundations of sequential digital circuits in this case.

Our prototype makes essential use of *Arrows*, a programming abstraction due to Hughes (2000), to capture what the various components can observe about the system. We motivate their use in Chapters 5 and 6.

1.3 Synopsis

This thesis aims to convince the reader of the practicality of mechanically reasoning about knowledge when designing certain kinds of systems. We seek to do this by providing a theory of the implementation of knowledge-based programs, building a prototype implementation using novel functional programming techniques, and applying it to several examples. While many of the examples are standard, they have not been treated in this generality before.

This thesis therefore reports on two threads of work:

- An automata-theoretic algorithm for replacing knowledge conditionals with concrete tests; and
- Embedded domain-specific languages (EDSLs) for control- and data-oriented circuits with knowledge conditionals.

The first part presents the theory.

Chapter 2: Reasoning about knowledge. We review mechanically reasoning about knowledge using modal logic. The existing tools closest to our domain are the epistemic model checkers, and in particular we discuss our experience with MCK (Gammie and van der Meyden 2004), a prototype symbolic model checker that supports temporal and knowledge modalities with a variety of semantics. Our implementation adopts several techniques developed for this tool. We also further motivate the KBP formalism and discuss what it means to implement a KBP.

Chapter 3: A theory of knowledge-based programs in Isabelle/HOL. We formalise a theory of KBPs and develop an algorithm for finding implementations under certain conditions. This algorithm is constructed in such a way that the code generator in Isabelle/HOL can be used to execute it. We apply the resulting program to two standard examples: the autonomous robot we discuss in Chapter 2 and the Muddy Children puzzle (Fagin et al. 1995, §1.1, Example 7.2.5).

This work was reported in Gammie (2011a) and Gammie (2011b).

The second part of this thesis discusses the implementation and applications.

Chapter 4: Synchronous digital circuits as functional programs. We review the long tradition of describing synchronous digital circuits as functional programs which our implementation builds upon. In particular we discuss the issues that arise when embedding a domain-specific language for circuits, and also some models closely related to the approach we adopt in the following chapter.

This chapter has been accepted for publication in the ACM Computing Surveys.

Chapter 5: Arrows for synchronous digital circuits. Here we discuss the foundations of our symbolic implementation of the theory of Chapter 3. We use *Arrows*, a functional programming abstraction due to Hughes (2000), and the *finally-tagless* representation championed by Carette, Kiselyov, and Shan (2009). We call the resulting embedded language ADHOC. We also discuss a control-oriented *Arrow transformer* which owes its foundations to Berry's seminal Esterel synchronous programming language (Berry 1999b). This layer was christened Kesterel by Kai Engelhardt.

Chapter 6: Knowledge-based circuits and applications. In this chapter we add constructs for knowledge-based programming to the circuit Arrows. These rely on the ability of Arrows to finely control information flow. We give symbolic versions of the algorithms developed in Chapter 3, and treat a series of standard examples more fully and automatically than they have been by previous systems. We conclude with a study of cache coherence that makes full use of the techniques developed in this thesis.

Chapter 7: Conclusions and future work. The final chapter summarises our contributions and discusses some of the outstanding issues. In particular we review the role that Arrows play in this work, suggest alternative implementation techniques and how the formalism of knowledge-based programming used here might usefully be extended.

There are also two appendices of related work.

Appendix A: Model Checking Knowledge and Linear Time: PSPACE Cases. Verifying the epistemic properties of standard systems has some similarities with the process of constructing implementations of KBPs. Here we establish some new results on the computational complexity of some model checking problems involving knowledge and linear time.

This work was reported in Engelhardt, Gammie, and van der Meyden (2007).

Appendix B: The Worker/Wrapper Transformation. This technique was developed by Gill and Hutton (2009) to change "a computation of one type into a worker of a different type, together with a wrapper that acts as an impedance matcher between the original and new computations." It was used to "semi-formally" refine circuits described in a language embedded into Haskell (§4.2.6) by Gill and Farmer (2011). This appendix presents a mechanisation of the foundational results, a corrected fusion rule with a correctness proof, and a new example.

This work was reported in Gammie (2009) and Gammie (2011c).

1.4 How to read this thesis

The two main artifacts described in this document are better experienced on a computer than on paper.

The proofs in Chapter 3 provide only an overview; full details are available in the KBP entry in the Archive of Formal Proofs (Gammie 2011a). Similarly further examples of the worker/wrapper transformation of Appendix B can be found there (Gammie 2009).

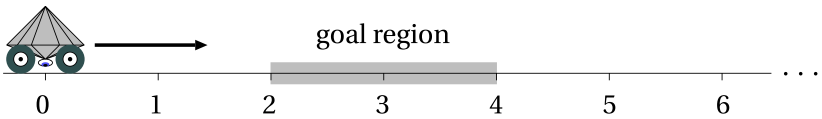
The implementation of the symbolic algorithm discussed in Chapters 5 and 6 is available from:

`http://peteg.org/circuits-and-knowledge/`

Chapter 2

Reasoning about knowledge

IMAGINE a robot stranded at zero on a discrete number line, hoping to reach and remain in the goal region $\{2,3,4\}$. The environment helpfully pushes the robot to the right, zero or one steps per unit time, and the robot can sense the current position with an error of plus or minus one. If the only action the robot can take is to halt at its current position, what program should it execute?



(image by Kai Engelhardt, tgif on Linux, 2001)

An intuitive way to specify the robot's behaviour is with a *knowledge-based program* (KBP), using the syntax of Dijkstra's guarded commands:

$$
\begin{array}{l}
\textbf{do} \\
\quad [\!]\ \mathbf{K}_{\mathsf{robot}}\,\mathsf{goal} \quad \rightarrow \mathsf{Halt} \\
\quad [\!]\ \neg\mathbf{K}_{\mathsf{robot}}\,\mathsf{goal} \rightarrow \mathsf{Nothing} \\
\textbf{od}
\end{array}
$$

where "$\mathbf{K}_{\mathsf{robot}}\,\mathsf{goal}$" intuitively denotes "the robot knows it is in the goal region" (Fagin et al. 1995, Example 7.2.2). We will make this more precise in §2.1, but for now note that what the robot knows depends on the rest of the scenario, which in general may involve other agents also running KBPs. It has been argued at length elsewhere (Fagin et al. 1995, Chapter 7) that this is a useful level of abstraction at which to reason about distributed systems, and some kinds of multi-agent systems (Shoham and Leyton-Brown 2008). The downside is that these specifications are not directly executable, and it may take significant effort to find a concrete program that has the required behaviour. We discuss the issues further in §2.2.

At this point the reader might like to try to find a predicate of the sensor reading that guarantees the robot will halt in the goal region. We could then verify that such a solution is in fact correct using an epistemic model checker. This approach has some subtleties that we discuss in §2.4 after reviewing the standard account of reasoning about knowledge using modal logic in the next section, and these tools in §2.3. The technology discussed there also features in our approach to automatically finding implementations of KBPs that we recount in Chapters 3 and 6.

2.1 Modal logics of knowledge

At the centre of many modern accounts of knowledge is modal logic, which has a long and venerable tradition too broad and deep to recount here; see Chellas (1980) for a general overview, and Hintikka (1962) and Fagin et al. (1995) for book-length treatments of its application to epistemics in particular.

Our logic has this syntax:

$$\phi, \psi ::= p \mid \neg\phi \mid \phi \wedge \psi \mid \mathbf{K}_a\phi$$

where we add a *knowledge modality* $\mathbf{K}_a\phi$ to the familiar infrastructure of classical propositional logic, with p ranging over propositions and a over agent identifiers. Intuitively $\mathbf{K}_a\phi$ means that agent a knows that ϕ holds.

For our purposes we can directly interpret this language into a *Kripke structure*, consisting of a non-empty set of *possible worlds* and one *accessibility relation* between these worlds for each agent a, written $w \sim_a w'$ for worlds w and w'. Finally a *valuation function* indicates the truth of a proposition at a world. Satisfaction of a formula ϕ at world w of structure M is defined as follows:

$$\begin{array}{ll}
M, w \models p & \text{iff } p \text{ is true at } w \text{ in } M \\
M, w \models \neg\phi & \text{iff } M, w \models \phi \text{ is false} \\
M, w \models \phi \wedge \psi & \text{iff } M, w \models \phi \text{ and } M, w \models \psi \\
M, w \models \mathbf{K}_a\phi & \text{iff } M, w' \models \phi \text{ for all worlds } w' \text{ where } w \sim_a w' \text{ in } M
\end{array}$$

What makes this a story about knowledge is the additional restriction we place on the accessibility relations: each must be an equivalence. Intuitively, for agent a to know ϕ at w (i.e., for $M, w \models \mathbf{K}_a\phi$ to hold) is for the formula to be true at all worlds in M that a cannot distinguish from w. These models are called $\mathrm{S5}_n$ structures after the axioms they satisfy.

Recalling our robot example, we can illustrate these definitions by considering that, from the robot's perspective, the positions $\{2,3,4\}$ are indistinguishable given a sensor reading of 3; therefore the relation for the robot would consider the worlds representing those positions equivalent. However if the robot knows how much time has elapsed then it may be able to reduce its uncertainty; for instance, if just two instants have passed and the sensor reads 3 then the robot can infer it is at position 2. This implies we should not rush to identify the worlds in the model M with the states of the system; we expand on this point in §3.4 after formally developing the concepts of modal logic we need in §3.2.

This semantics assumes that the agents are very powerful reasoners; in particular it expects them to be *logically omniscient*, to be able to infer any conclusion that is sound with respect to what they know. While this is clearly fallacious if we wish to reason about social or computationally bounded agents, we take the perspective of knowledge as a design tool: from outside the system we talk about agents knowing things while inside the system we look for concrete ways for the agents to behave as if they did know those things. This is the implementation problem we discuss further in §2.2. We note that this is in direct opposition to the kind of assumptions typically made about cryptographic primitives.

These and many other philosophical issues are canvassed at length by Lenzen (1978) for both knowledge and belief.

The next section discusses how an extension of this logic can be used to describe how communication amongst the agents affects their epistemic state, provided the facts themselves do not change.

2.1.1 Dynamic epistemic logic

The field of *dynamic logic* is a way of analysing change by studying the effect of actions, using formulas to describe both actions and states of the world. Traditionally it is considered a generalisation of Floyd-Hoare logic, an approach to giving programming language semantics axiomatically that is widely used in the program verification community (Winskel 1993, Chapter 6). van Eijck and Stokhof (2006) provide a historical overview of this field with coverage of epistemic applications.

In *dynamic epistemic logic* (DEL) the actions are communication events, and the change being studied is that of the epistemic state of the agents, while the scenario ("the facts") remains constant (Baltag and Moss 2004; Plaza 2007).

As our focus here is on algorithmically reasoning about epistemic situations, we restrict our attention to the only sublogic that has been mechanised: *public announcement logic* (PAL). There arbitrary formulas of the logic are broadcast, as if from a loudspeaker within the (simultaneous) hearing of all agents. The syntax is as for our modal account of the previous section augmented with a "dynamic" modality:

$$\phi, \psi ::= p \mid \neg\phi \mid \phi \wedge \psi \mid \mathbf{K}_a\phi \mid [\phi]\psi$$

Intuitively $[\phi]\psi$ means that after ϕ has been publicly announced, ψ holds. Formally the semantics for this language is as before with the extra clause:

$$M, w \models [\phi]\psi \ \text{ iff } \ M, w \models \phi \text{ implies } M|\phi, w \models \psi$$

where the restriction operator $M|\phi$ defines the Kripke structure consisting of worlds $W' = \{w' \in W \mid M, w' \models \phi\}$, with the relations and valuation restricted to W'.

Note that iterating the update operator effectively gives the agents *perfect recall*; the agents do not forget what they have learnt from previous broadcasts. We provide our knowledge-based programs with similar powers in similar settings, as we discuss in §3.7.2.

PAL as presented here cannot describe our robot example as "the facts" can change at every time step. ("The facts" there include the robot's position and sensor reading, and perhaps whether it has halted.) There has been some work towards rectifying this by integrating temporal notions with dynamic epistemic logic; van Ditmarsch, van der Hoek, and Ruan (2011) review the situation, but at present there is no tool support.

The following section discusses DEMO, which implements PAL as presented here.

DEMO

DEMO (Dynamic Epistemic MOdelling) is a tool written by van Eijck (2007) in Haskell (Peyton Jones 2003) that implements the logic of Baltag and Moss (2004). It provides a way of describing a multi-agent epistemic scenario with static concrete facts by giving a multi-modal Kripke structure which can be updated by public and private announcements. Note that these operations can destroy the property that the relations are equivalences, and hence the resulting structures may no longer capture some notion of knowledge. DEMO cannot treat our robot example as it lacks the ability to handle factual change.

van Ditmarsch, Ruan, and Verbrugge (2008) provide an overview of this tool and show how it can be used to solve the product and sum puzzle which we treat in §6.4.2 as a knowledge-based program.

DEMO can be seen as an *embedded domain-specific language* (EDSL), a library which effectively extends the host language (Haskell in this case) with constructs that are specifically designed for modelling epistemic scenarios. We adopt a similar approach in our work and discuss the issues at length in Chapters 4 and 5.

The present implementation of DEMO is explicit-state and algorithmically naïve in some aspects. It does not attempt to address the state-explosion problem that plagues tools that reason about large systems (see Clarke et al. (1999, Chapter 1) and Valmari (1996) for more on this issue).

2.2 Knowledge-based programs

The knowledge-based program formalism extends the modal account of knowledge with dynamism in a distinct manner to DEL: *protocols* (sets of guarded commands) are used to express the explicit dependency of action on knowledge. The original motivation was to provide a more abstract way of specifying protocols for distributed systems that concords more closely with intuitions about their correctness than other techniques, and also a means of talking about optimal use of information which is useful for minimising communication. It has been applied to several examples, such as the bit transmission problem (Halpern and Zuck 1992) and cache coherence (Baukus and van der Meyden 2004), with the vast majority being small enough to yield to pencil-and-paper efforts. Our goal is to provide mechanical assistance so that we may treat larger examples.

Our first difficulty is a semantic one: while we can give meaning to the actions of a KBP in a standard way, we need a Kripke structure to interpret the guards. What is the provenance of this structure?

Intuitively the guards in a set of KBPs should be interpreted with respect to the very set of traces (sequences of global states) generated by their execution, for then we have a justification for all agent actions, and moreover we do not consider spurious traces of the system that might interfere with (and weaken) the epistemic states of the agents. Thus the most general semantics we can give KBPs involves a non-constructive fixed point that may have zero, one or many solutions; see §3.5 for a formal account.

The situation can be improved by including some notion of causality; for instance Fagin et al. (1995, §7.2) require that if a guard $\mathbf{K}_a\phi$ for agent a is false at time n on a world w, then there is a world w' in the structure that has occurred at time n or before where ϕ is actually false and $w \sim_a w'$. In other words, that an agent doesn't know something at time n can be justified by its observations of the system up to time n. This "provides witnesses" property guarantees that the set of traces generated by a KBP in a scenario is unique, and moreover can be constructed inductively (Fagin et al. 1995, Theorem 7.2.4). It applies to the common types of communication used in concurrent systems.

As we aspire to algorithmically construct finite-state representations of this set of traces (i.e., automata), assuming the environment they run in consists of a finite collection of states, we must further limit our horizons. (We argued in §1.2 that it is reasonable to ask for finite-state *implementations* of controllers and protocols that are intended to run forever.) Whether this is possible depends on what exactly the worlds of our Kripke structure are. One option is to use the set of traces themselves, with the indistinguishability relations being the lifting of the agents' instantaneous observations on global states to traces; this is termed the *synchronous perfect-recall view*, which clearly "provides witnesses" as the only worlds relevant to an agent's knowledge at time n are those that occur precisely at time n. In this case the answer is in the negative in general, but there are some special cases where it is possible (van der Meyden 1996b). Moreover implementations always exist with respect to the weaker *clock view* where agents observe the passing of time as well as some part of the global system state. This motivates the fragmentary approach of the next chapter, where we discuss the semantics in more detail.

An alternative approach is to verify that a proposed implementation does satisfy the fixed point. We discuss the subtleties of this approach in the following section.

The KBP formalism is a partial answer to the concerns Wolper (1998) has about synthesis of concurrent systems from formulas of a temporal logic: there the resulting artifacts are centralised solutions. Here we build the communication structure of the solution into the scenario description, using the notion of an agent. It seems natural to require the agents' observations and possible actions to be specified, though we grant the KBP formalism may be overly restrictive in requiring knowledge and action to be so rigidly coupled.

The problem we consider in this thesis is subsumed by the synthesis problem for knowledge-based *specifications* (van der Meyden and Vardi 1998), which involves determining what sequence of actions will lead to a temporal and epistemic objective. That problem is clearly a lot more computationally complex.

2.3 Model-checking knowledge

Model checkers are fully automatic tools for verifying that statements in particular logics hold in specific models. Development of such tools for epistemic logic was driven by a general interest in analysing systems for properties such as anonymity, as provided by e.g. the Dining Cryptographers protocol (Chaum 1988), which we say more about in §6.5. From the perspective of finding implementations of knowledge-based programs, we review these model checkers for two reasons: firstly, as we mentioned in §2.2 we can use them to verify the implementation relation, and secondly we use their underlying technology in our implementation in Chapter 6. Simply put, these are the systems most closely related to ours.

All of the tools covered in this section extend a temporal logic with epistemic modalities. Temporal logic comes in two basic kinds: *computation tree logic* (CTL), a branching time logic, and *linear temporal logic* (LTL). As we make no deep use of these logics we defer the details to the canonical reference by Clarke et al. (1999, Chapter 3) and explain the little we need as we go along.

The initial approaches to automatic verification of epistemic properties, such as van der Hoek and Wooldridge (2003), involved a translation into an existing tool-supported temporal logic. However these translations require some manual analysis of the epistemic property of interest – for instance it is *a priori* unclear that a perfect recall semantics for knowledge can be so encoded, and hence just how strong an anonymity claim is. As these give us no insight into our algorithmic task, we omit coverage of them. Similarly we ignore the more exotic modal operators such as common and distributed knowledge. These are typically easy to add to the core semantics.

We note that model-checking knowledge in combination with time can be computationally complex. The basic problem of checking the satisfiability of a knowledge formula in a Kripke structure is linear in the size of the formula and number of worlds (Fagin et al. 1995, Proposition 3.2.1), but in concert with time and more sophisticated semantics for knowledge the complexity increases significantly; see Appendix A. This has given rise to many approaches and special cases.

Model checkers have traditionally used one of three basic technologies: explicit traversal of state spaces, and symbolic encodings into Boolean decision diagrams (BDDs) or propositional satisfiability (SAT). We treat each separately in the following sections, and comment further on their pragmatics in §6.5.

2.3.1 Explicit-state model checking

The original tools for verification of finite-state concurrent systems used an explicit representation of the state space. The venerable exemplar of this technique is SPIN (Holzmann 1997), which uses (sublanguages of) LTL for property specifications. The issue of state-space explosion is treated via sophisticated techniques such as partial-order reduction and predicate abstraction.

To our knowledge DEMO (§2.1.1) is the only epistemic tool that uses this approach.

2.3.2 Model checking using Boolean decision diagrams

Boolean decision diagrams (BDDs) canonically represent classical Boolean functions of type $\mathbb{B}^n \rightarrow \mathbb{B}$, and also support quantification over Boolean variables. As such they can solve the canonical PSPACE-complete problem of determining the satisfiability of quantified Boolean formulas (QBFs). They underpin a natural algorithm for checking that a CTL formula is satisfiable in a finite model.

Abstractly a BDD is a binary tree with internal nodes labelled with Boolean variables and leaves with the constants true or false. Each path from the root to a leaf encodes a partial assignment of the variables, where the variable is given complementary values in its two descendant subtrees. A leaf indicates the value of the function on the set of variable assignments represented by the path leading to it. By sharing subtrees BDDs can potentially economise over full truth tables.

The canonicity of BDDs arises from imposing a global ordering on the variables and requiring that it be respected by all paths in all BDDs. With a global cache that uniquely records all BDDs in existence, we can test whether two BDDs are equal with a simple pointer equality check. This property makes it easy to determine the stopping condition of the fixed point computations which underpin reachability and CTL model checking algorithms. As the size of a BDD depends critically on this ordering, and the BDD minimisation problem is NP-complete, modern BDD packages provide heuristics that search for a permutation of this order that reduces space consumption. This means that it is often very difficult to evaluate the quality of BDD-based algorithms; we return to this point in §6.5.

As the literature on these data structures is vast, and we will use them substantially as an abstract data type, we defer further background to Clarke et al. (1999, Chapter 6).

The first application of BDDs to general-purpose epistemic model checking was in MCK (Gammie and van der Meyden 2004), which has a variety of combinations of temporal logics and semantics for knowledge that we sketch below. As with all other tools apart from DEMO, it can verify epistemic conditions where “the facts” change with time. It is written in Haskell (Peyton Jones 2003).

MCK requires that a description of a multi-agent scenario specify the observations made by each agent with a projection of the instantaneous global state of the system. (In terms of the *interpreted systems* framework of Fagin et al. (1995), we construct agents’ *local states* using this observation. See §3.4.) The *observational* semantics for knowledge uses a Kripke structure where the worlds consist of the set of all reachable states, and each agent’s equivalence relation is induced by their observation function. It is then straightforward to incorporate this semantics with the standard accounts of temporal logic; knowledge subformulas involve a quantification over the set of all reachable states that are indistinguishable from the final state on a trace.

This is the weakest semantics for knowledge as it does not record the history of the agents’ observations. We have more to say about this in §2.4.

At the other end of the spectrum is the *perfect recall* semantics for knowledge. There the worlds of the Kripke structure are the traces of the system and the equivalence relations arise from the pointwise lifting of the agents’ observation functions. (See §3.7.2 for the formal details, and Baukus and van der Meyden (2004) for details of the *asynchronous* perfect-recall semantics.) MCK supports several limited combinations of time and knowledge in this case as these are quite computationally complex in general (see van der Meyden and Shilov (1999) and Appendix A).

For checking invariant knowledge properties with respect to perfect recall, MCK uses an algorithm similar to that presented in the next chapter. It also uses a simplified construction that is similar to the bounded model checking of LTL (Biere, Cimatti, Clarke, and Zhu 1999) for temporally-bounded formulas. Roughly, the latter involves unfolding the transition relation k times for formulas with the temporal next-state modality nested k times. The resulting BDD can be universally quantified over to determine which traces of length k are observationally identical for a given agent, which is the key to evaluating the semantics of the knowledge modality.

Perfect recall is very useful in showing strong ignorance properties, such as verifying that the Dining Cryptographers provides anonymity to its participants. The combination of this semantics with temporal logic is unique to MCK.

Of strength intermediate between the observational and perfect recall semantics is the *clock* semantics which incorporates a global clock with the agents’ observations. It has the advantage of being relatively computationally unimposing and general; see §3.7.1 for further details.

MCK has also been used to verify a solution to the Russian Cards problem (van Ditmarsch, van der Hoek, van der Meyden, and Ruan 2006), and also to check that particular standard and knowledge-based programs have identical behaviour as we discuss in §2.4.

The approximately contemporaneous system MCTK by Su, Sattar, and Luo (2007) implements a combination of LTL and the observational semantics by extending the BDD-based model checker NuSMV, which is written in C. In particular, they make use of the LTL-to-CTL translation due to Clarke, Grumberg, and Hamaguchi (1997) already implemented in NuSMV and also MCK.

MCTK has been used to verify the Dining Cryptographers protocol, a solution to the Russian Cards problem and the product and sum puzzle (Luo, Su, Sattar, and Chen 2008).

MCMAS (Lomuscio, Qu, and Raimondi 2009) is another model checker for multi-agent systems that employs BDDs. Its specification language is based on *alternating temporal logic* (ATL), which extends CTL with modalities for coalitions. The knowledge operator has the same meaning as it does for the observational semantics in MCK. MCMAS also has a deontic modality for expressing that an agent is complying with its protocol. The tool is implemented in C++, and has been used to verify the authentication properties of the TESLA protocol (Lomuscio, Raimondi, and Wozna 2007) in addition to the standard examples.

2.3.3 Model checking using SAT

Solvers for the canonical NP-complete Boolean satisfiability problem (SAT) have increased significantly in performance over the past decade to the point where they can routinely handle quite large instances in reasonable resource bounds. They have been successfully used in bounded model checkers (BMCs) for LTL (Biere et al. 1999). The tool VerICS by Kacprzak, Nabialek, Niewiadomski, Penczek, Pólrola, Szreter, Wozna, and Zbrzezny (2008) combines a bounded semantics for CTL with the (bounded) observational semantics for knowledge. It is written in C++.

The basic idea of temporal BMC is as for MCK's specialised treatment of nested temporal next operators: for a bound k, unfold the system's transition relation k times, translate the temporal formula into a propositional formula and feed the composition to a SAT solver. The semantics VerICS assigns to the knowledge modality is reminiscent of the "provides witnesses" condition from §2.2: Define $\bar{\mathbf{K}}_a\phi \equiv \neg\mathbf{K}_a\neg\phi$ ("a believes ϕ is possible") and give it the semantics:

$$M, w \models \bar{\mathbf{K}}_a\phi \text{ iff there exists } w' \text{ in } M \text{ such that } w \sim_a w' \text{ and } M, w' \models \phi$$

for a Kripke structure M where the worlds are the states reachable within k steps of an initial state, and the relations are induced by the agents' observation functions as for MCK's observational semantics. For large enough k this approach is equivalent to unbounded model checking using the observational semantics.

VerICS has been applied to the verification of the Dining Cryptographers protocol.

2.4 Verifying KBP implementations by model checking

As we have previously observed, one way to find implementations of knowledge-based programs is to propose a solution and verify that it has the desired epistemic properties. This is essentially a partial mechanisation of the pencil-and-paper approach used by Fagin et al. (1995, Chapter 7). We now discuss why using an epistemic model checker for this purpose requires some care.

Recall the robot of §2. A typical first stab at an implementation is to propose that the robot halt iff the sensor reads 3, as this is the only reading at which it is certain that it is in the goal region. We can indeed verify that $\mathbf{K}_{\mathsf{robot}}\ \mathsf{goal}$ iff the sensor reads 3 using the observational semantics for knowledge (see §2.3.2), which seems to justify replacing the knowledge test with the concrete one. Unfortunately there are runs of the system where the sensor never reads 3, and so the robot can sail straight through the goal region with this implementation.

With some further thought we can refine the concrete test to "the sensor reads *at least* 3" and again show that this is equivalent to the knowledge test using the observational semantics. Furthermore we can show that this concrete tests satisfies the liveness condition that the robot always halts, subject to the fairness condition that the environment infinitely often tries to move the robot to the right unless it has halted.

This issue of non-unique implementations can be rectified by assuming that the system "provides witnesses" Fagin et al. (1995, Theorem 7.2.4), as we have mentioned before. The clock semantics mentioned in the previous section is the weakest semantics for knowledge considered here that satisfies this assumption; clearly it is not satisfied by the observational semantics.

The second issue with the observational semantics is that it cannot be used to show that an agent is making optimal use of its information. For instance, the two applications of MCK in this methodology – a cache coherence protocol (Baukus and van der Meyden 2004) (see §6.6) and extensions of the Dining Cryptographers protocol (Al-Bataineh and van der Meyden 2010) – make essential use of the counter examples generated by the perfect recall semantics for knowledge to identify optimisation opportunities.

We note that that guess-and-check approach can potentially address the full range of KBPs, including those with an infinity of states and guards that use future-time temporal logic. We note that as knowledge is a property of the entire system it resists the development of compositional techniques, though more structure can be brought to the search for implementations with the refinement calculus developed by Engelhardt et al. (2001).

In contrast the tool we develop here is fully automatic.

2.5 Concluding remarks

The epistemic model checkers we considered above provide decision procedures on their domains of interest. A more powerful but less automatic technique involves proving theorems in an expressive logic, much as we use to demonstrate the meta-theory of KBPs in the next chapter.

McCarthy (1987) advocates using first-order logic to model knowledge. He makes the point that people mint modalities as needed and hence modal logic as traditionally formulated is too rigid for the purposes of general artificial intelligence. We contrast his treatment of the classic product and sum puzzle with a KBP-based approach in §6.4.2. A somewhat similar approach, again using theorem proving rather than model-based reasoning, is the *situation* *theory* of Barwise and Perry; see Ersan and Akman (1995) for an overview and application to some familiar epistemic puzzles. More recently Bickford et al. (2009) have extended *event theory* with knowledge operators using the proof assistant Nuprl.

Knowledge-based programs have also been used to reason about unobservable state in *discrete-event systems* by Ricker and Rudie (2007). The goal in this setting is to construct a controller that constrains the behaviour of a *plant* (system) by disabling certain controllable actions. In some cases it is desirable for the controller to be decentralised, which is topic that has been further explored by Bensalem, Peled, and Sifakis (2010) using knowledge-based techniques.

We discuss the robot example introduced in §2 further in §3.8.1 and §6.3, and other examples in Chapter 6. A two-dimensional variant of it was treated manually by Rosenschein and Kaelbling (1986) using a predicate logic. Their objective was to establish the soundness of a test for knowledge, but not its completeness. Moreover in their model action is not entirely determined by knowledge – they assume the robot has a policy for ambling about, but their implementation of the knowledge conditionals is independent of it.

Brafman, Latombe, Moses, and Shoham (1997) use a knowledge-based approach to treat the problem of motion planning under uncertainty in greater generality, using this robot example as an intuition pump.

Alternative semantics for KBPs have been proposed, one involving predicate transformers (Sanders 1991), and another dynamic epistemic logic (de Haan, Hesselink, and de Lavalette 2004).

Chapter 3

A theory of knowledge-based programs in Isabelle/HOL

THERE are several subtleties in deriving concrete programs that implement knowledge-based programs, with characterising just what "implement" means being one of the first. This chapter presents a mechanised proof of correctness for a particular algorithmic approach to this task, building on the work of Fagin et al. (1995, Chapter 7) and van der Meyden (1996b).

3.1 Proof overview

The objectives of this development are to design a generic algorithm that automatically constructs implementations of KBPs, specialise it to particular semantics for knowledge, and run it on two standard scenarios.

We use Isabelle, a mature logical framework that hosts an implementation of the *simply-typed higher-order logic* (HOL) originated by Church and popularised by Mike Gordon and his colleagues; Nipkow, Paulson, and Wenzel (2002) provide a tutorial introduction.

The development is top-down, and proceeds as follows:

§3.2 formally defines the syntax and Kripke semantics of our particular logic of knowledge, and the auxiliary concepts of sub-models of, and simulations between Kripke structures.

§3.3 introduces knowledge-based programs and gives a semantics with respect to Kripke structures, and shows it to be invariant under certain relations amongst such structures.

§3.4 models the environments of interest, their traces and how agents perceive these environments through *views*. These underpin a semantics for KBPs where we interpret the knowledge conditionals with respect to a given set of traces.

§3.5 shows that assuming views to be synchronous yields a canonical, inductively constructed set of traces for the KBPs.

§3.6 defines a class of automata, shows what it means to implement a set of KBPs and develops an algorithm that inductively constructs these automata.

§3.7 instantiates the algorithm for the *clock* and *synchronous perfect recall* views.

§3.8 applies this machinery to two examples: the Robot from Chapter 2, and the classic Muddy Children puzzle.

The following sections are rendered directly from the Isabelle/HOL proofs, and use notation quite similar to mathematical convention. We make extensive use of Isabelle's *locales* (Ballarin 2006; Kammüller, Wenzel, and Paulson 1999) which provide a convenient way of stating a series of lemmas relative to a fixed context, and also of extending and instantiating these contexts. They are an approximate logical equivalent of the *functors* of Standard ML.

As the reader can find the complete development in the Archive of Formal Proof (Gammie 2011a), we suppress many details. Those impatient to see the final result can find the algorithm in Figure 3.4 on page 41, and the definitions for the Robot of §2 in Figure 3.8 on page 70.

3.2 A modal logic of knowledge

We begin with the standard syntax and semantics of the propositional logic of knowledge based on *Kripke structures*. More extensive treatments can be found in Lenzen (1978), Chellas (1980), Hintikka (1962) and Fagin et al. (1995, Chapter 2).

The syntax includes one knowledge modality per agent, and one for *common knowledge* amongst a set of agents. It is parameterised by the type `'a` of agents and `'p` of propositions.

```
datatype ('a, 'p) Kform
  = Kprop "'p"
  | Knot "('a, 'p) Kform"
  | Kand "('a, 'p) Kform" "('a, 'p) Kform"
  | Kknows "'a" "('a, 'p) Kform"  ("K_ _")
  | Kcknows "'a list" "('a, 'p) Kform" ("C_ _")
```

A Kripke structure consists of a set of *worlds* of type `'w`, one *accessibility relation* between worlds for each agent and a *valuation function* that indicates the truth of a proposition at a world. This is a very general story that we will quickly specialise.

```
record ('a, 'p, 'w) KripkeStructure =
  worlds :: "'w set"
  relations :: "'a ⇒ ('w × 'w) set"
  valuation :: "'w ⇒ 'p ⇒ bool"
```

```
definition kripke :: "('a, 'p, 'w) KripkeStructure ⇒ bool" where
  "kripke M ≡ ∀a. relations M a ⊆ worlds M × worlds M"
```

```
definition mkKripke :: "'w set ⇒ ('a ⇒ ('w × 'w) set) ⇒ ('w ⇒ 'p ⇒ bool)
                         ⇒ ('a, 'p, 'w) KripkeStructure" where
  "mkKripke ws rels val ≡
     (| worlds = ws, relations = λa. rels a ∩ ws × ws, valuation = val |)"
```

The standard semantics for knowledge is given by taking the accessibility relations to be equivalence relations, yielding the S5$_n$ structures, so-called due to their axiomatisation.

```
definition S5n :: "('a, 'p, 'w) KripkeStructure ⇒ bool" where
  "S5n M ≡ ∀a. equiv (worlds M) (relations M a)"
```

3.2.1 Satisfaction

A formula ϕ is satisfied at a world w in Kripke structure M in the following way:

```
fun models :: "('a, 'p, 'w) KripkeStructure ⇒ 'w ⇒ ('a, 'p) Kform
           ⇒ bool" ("(_, _ ⊨ _)" [80,0,80] 80) where
  "M, w ⊨ (Kprop p) = valuation M w p"
| "M, w ⊨ (Knot φ) = (¬ M, w ⊨ φ)"
| "M, w ⊨ (Kand φ ψ) = (M, w ⊨ φ ∧ M, w ⊨ ψ)"
| "M, w ⊨ (K_a φ) = (∀w' ∈ relations M a `` {w}. M, w' ⊨ φ)"
| "M, w ⊨ (C_as φ) = (∀w' ∈ (⋃a ∈ set as. relations M a)+ `` {w}. M, w' ⊨ φ)"
```

The first three clauses interpret propositional logic in the standard way.

The clause for $\mathtt{K_a}\ \varphi$ expresses the idea that an agent knows φ at world `w` in structure `M` iff φ is true at all worlds it considers possible.

The clause for $\mathtt{C_{as}}\ \varphi$ captures what it means for the set of agents `as` to commonly know φ; roughly, all agents in `as` know φ, and know that all members of `as` know it, and so forth. Note that the transitive closure and the reflexive-transitive closure generate the same relation due to the reflexivity of the agents' accessibility relations.

The relation between knowledge and common knowledge can be understood as follows, following Fagin et al. (1995, §2.4). Firstly, that φ is common knowledge to a set of agents `as` can be seen as asserting that all agents in `as` know φ and moreover know that it is common knowledge amongst `as`.

```
lemma S5n_common_knowledge_fixed_point:
  assumes "S5n M"
  assumes "w ∈ worlds M"
  assumes "a ∈ set as"
  shows "M, w ⊨ Kcknows as φ
     ⟷ M, w ⊨ Kand (Kknows a φ) (Kknows a (Kcknows as φ))"
```

Secondly we can provide an induction schema for the introduction of common knowledge: from all agents in `as` knowing that φ implies $\varphi \land \psi$, and that φ is satisfied at world `w`, infer that ψ is common knowledge amongst `as` at `w`.

```
lemma S5n_common_knowledge_induct:
  assumes S5n: "S5n M"
  assumes w: "w ∈ worlds M"
  assumes E: "∀a ∈ set as. ∀w ∈ worlds M.
                  M, w ⊨ φ ⟶ M, w ⊨ K_a (Kand φ ψ)"
  assumes p: "M, w ⊨ φ"
  shows "M, w ⊨ C_as ψ"
```

The rest of this section introduces the technical machinery we use to relate Kripke structures.

3.2.2 Generated models

Intuitively the truth of a formula at a world depends only on the worlds that are reachable from it in zero or more steps, using any of the accessibility relations at each step. Traditionally this result is called the *generated model property* (Chellas 1980, §3.4). Concretely we generate a (sub-)model of `M` from `w` by taking the image of `w` under the reflexive transitive closure of the agents' relations as follows:

```
definition
  gen_model :: "('a, 'p, 'w) KripkeStructure ⇒ 'w ⇒ ('a, 'p, 'w) KripkeStructure"
where
  "gen_model M w ≡
    let ws' = worlds M ∩ (⋃a. relations M a)* `` {w}
     in ⦇ worlds = ws',
          relations = λa. relations M a ∩ (ws' × ws'),
          valuation = valuation M ⦈"
```

We can show that the satisfaction of a formula φ at a world `w'` is preserved, provided `w'` is relevant to the world `w` that the sub-model is based upon, by structural induction over φ.

```
lemma gen_model_semantic_equivalence:
  assumes M: "kripke M"
  assumes w': "w' ∈ worlds (gen_model M w)"
  shows "M, w' ⊨ φ ⟷ (gen_model M w), w' ⊨ φ"
```

3.2.3 Simulations

A *simulation*, or *p-morphism*, is a mapping from the worlds of one Kripke structure to another that preserves the truth of all formulas at related worlds (Chellas 1980, §3.4, Ex. 3.60). Such a function `f` must satisfy four properties. Firstly, the image of the set of worlds of `M` under `f` should equal the set of worlds of `M'`.

```
definition sim_range :: "('a, 'p, 'w1) KripkeStructure
                       ⇒ ('a, 'p, 'w2) KripkeStructure ⇒ ('w1 ⇒ 'w2) ⇒ bool"
where "sim_range M M' f ≡ worlds M' = f ` worlds M
                          ∧ (∀a. relations M' a ⊆ worlds M' × worlds M')"
```

The value of a proposition should be the same at corresponding worlds:

```
definition sim_val :: "('a, 'p, 'w1) KripkeStructure
                      ⇒ ('a, 'p, 'w2) KripkeStructure ⇒ ('w1 ⇒ 'w2) ⇒ bool"
where "sim_val M M' f ≡ ∀u ∈ worlds M. valuation M u = valuation M' (f u)"
```

If two worlds are related in `M`, then the simulation maps them to related worlds in `M'`; intuitively the simulation relates enough worlds. We term this the *forward* property.

```
definition sim_f :: "('a, 'p, 'w1) KripkeStructure
                    ⇒ ('a, 'p, 'w2) KripkeStructure ⇒ ('w1 ⇒ 'w2) ⇒ bool"
where "sim_f M M' f ≡ ∀a u v. (u, v) ∈ relations M a ⟶ (f u, f v) ∈ relations M' a"
```

Conversely, if two worlds `f u` and `v'` are related in `M'`, then there is a pair of related worlds `u` and `v` in `M` where `f v = v'`; intuitively the simulation makes enough distinctions. We term this the *reverse* property.

```
definition sim_r :: "('a, 'p, 'w1) KripkeStructure
                    ⇒ ('a, 'p, 'w2) KripkeStructure ⇒ ('w1 ⇒ 'w2) ⇒ bool"
where "sim_r M M' f ≡ ∀a. ∀u ∈ worlds M. ∀v'.
            (f u, v') ∈ relations M' a ⟶ (∃v. (u, v) ∈ relations M a ∧ f v = v')"

definition "sim M M' f ≡ sim_range M M' f ∧ sim_val M M' f
                         ∧ sim_f M M' f ∧ sim_r M M' f"
```

Due to the common knowledge modality, we need to show the simulation properties lift through the transitive closure. In particular we can show that the forward and reverse simulation properties are preserved:

```
lemma sim_f_tc:
  assumes s: "sim M M' f"
  assumes uv': "(u, v) ∈ (⋃a∈as. relations M a)⁺"
  shows "(f u, f v) ∈ (⋃a∈as. relations M' a)⁺"

lemma sim_r_tc:
  assumes M: "kripke M"
  assumes s: "sim M M' f"
  assumes u: "u ∈ worlds M"
  assumes fuv': "(f u, v') ∈ (⋃a∈as. relations M' a)⁺"
  obtains v where "f v = v'" and "(u, v) ∈ (⋃a∈as. relations M a)⁺"
```

Finally we establish the key property of simulations, that they preserve the satisfaction of all formulas in the following way:

```
lemma sim_semantic_equivalence:
  assumes M: "kripke M"
  assumes s: "sim M M' f"
  assumes u: "u ∈ worlds M"
  shows "M, u ⊨ φ ⟷ M', f u ⊨ φ"
```

The proof is by structural induction over the formula φ. The knowledge cases appeal to our simulation preservation lemmas.

This is all we need to know about Kripke structures. Sangiorgi (2009) surveys p-morphisms and the related concept of *bisimulation* more broadly.

3.3 Knowledge-based programs

A knowledge-based program (KBP) encodes the dependency of action on knowledge by a sequence of guarded commands, and a *joint knowledge-based program* (JKBP) assigns a KBP to each agent:

```
record ('a, 'p, 'aAct) GC =
  guard  :: "('a, 'p) Kform"
  action :: "'aAct"

type_synonym ('a, 'p, 'aAct) KBP = "('a, 'p, 'aAct) GC list"
type_synonym ('a, 'p, 'aAct) JKBP = "'a ⇒ ('a, 'p, 'aAct) KBP"
```

We use a list of guarded commands just so we can reuse this definition and others in algorithmic contexts; we would otherwise use a set as there is no problem with infinite programs or actions, and we always ignore the sequential structure.

Intuitively a KBP for an agent cannot directly evaluate the truth of an arbitrary formula as it may depend on propositions that the agent has no certainty about. For example, a card-playing agent cannot determine which cards are in the deck, despite being sure that those in its hand are not. However agent a can certainly evaluate formulas of the form $K_a\ \varphi$ as these merely require φ to be true at all worlds that a considers possible. Therefore we restrict the guards of the JKBP to be boolean combinations of *subjective* formulas:

```
fun subjective :: "'a ⇒ ('a, 'p) Kform ⇒ bool" where
  "subjective a (Kprop p)   = False"
| "subjective a (Knot φ)    = subjective a φ"
| "subjective a (Kand φ ψ) = (subjective a φ ∧ subjective a ψ)"
| "subjective a (K_a' φ)    = (a = a')"
| "subjective a (C_as φ)    = (a ∈ set as)"
```

All JKBPs in the following sections are assumed to be subjective.

```
lemma S5n_subjective_eq:
  assumes S5n: "S5n M"
  assumes subj: "subjective a φ"
  assumes ww': "(w, w') ∈ relations M a"
  shows "M, w ⊨ φ ⟷ M, w' ⊨ φ"
```

The proof is by induction over the formula φ, using the properties of $S5_n$ Kripke structures in the knowledge cases.

We capture the fixed but arbitrary JKBP using a locale, and work in this context for the rest of this section.

```
locale JKBP =
  fixes jkbp :: "('a, 'p, 'aAct) JKBP"
  assumes subj: "∀a gc. gc ∈ set (jkbp a) ⟶ subjective a (guard gc)"

context JKBP
begin
```

The action of the JKBP at a world is the list of all actions that are enabled at that world:

```
definition jAction :: "('a, 'p, 'w) KripkeStructure ⇒ 'w ⇒ 'a ⇒ 'aAct list"
where "jAction ≡ λM w a. [ action gc. gc ← jkbp a, M, w ⊨ guard gc ]"
```

All of our machinery on Kripke structures from §3.2 lifts through jAction, due to the subjectivity requirement. In particular, the KBP for agent a behaves the same at worlds that a cannot distinguish amongst:

```
lemma S5n_jAction_eq:
  assumes S5n: "S5n M"
  assumes ww': "(w, w') ∈ relations M a"
  shows "jAction M w a = jAction M w' a"
```

Also the JKBP behaves the same on relevant generated models for all agents, and is invariant under simulations.

```
lemma gen_model_jAction_eq:
  assumes S: "gen_model M w = gen_model M' w"
  assumes w': "w' ∈ worlds (gen_model M' w)"
  assumes M: "kripke M" and M': "kripke M'"
  shows "jAction M w' = jAction M' w'"

lemma simulation_jAction_eq:
  assumes M: "kripke M"
  assumes sim: "sim M M' f"
  assumes w: "w ∈ worlds M"
  shows "jAction M w = jAction M' (f w)"

end
```

3.4 Environments and views

The previous section showed how a JKBP can be interpreted statically, with respect to a fixed Kripke structure. We capture how agents interact by adopting the *interpreted systems* and *contexts* of Fagin et al. (1995), which we term *environments* following van der Meyden (1996b).

We extend the JKBP locale with the following constants:

```
locale PreEnvironment = JKBP jkbp for jkbp :: "('a, 'p, 'aAct) JKBP"
+ fixes envInit :: "'s list"
    and envAction :: "'s ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's"
    and envVal :: "'s ⇒ 'p ⇒ bool"
```

A *pre-environment* is a JKBP and a description of its environment, which consists of an arbitrary set of initial states (envInit), the non-deterministic protocol of the environment envAction, which can depend on the current state, a transition function `envTrans` that composes the environment's action and agents' behaviour into a state change, and a propositional valuation function `envVal`. In general `envTrans` may incorporate a scheduler and communication failure models.

We represent the possible evolutions of the system as finite sequences of states, represented by a left-recursive type `'s Trace` with constructors `tInit s` and `t ⇝ s` for a trace `t` and state `s`, equipped with tFirst, tLast, tLength, tMap and tZip functions.

Constructing these traces requires us to determine the agents' actions at a given state, which in turn means we need an appropriate $S5_n$ structure for interpreting `jkbp`. Given that we want the agents to make optimal use of the information they have, we allow this structure to depend on the entire history of the system, suitably conditioned by what the agents can observe. We capture this notion of observation with a *view* (van der Meyden 1996b):

```
type_synonym ('s, 'tview) View = "'s Trace ⇒ 'tview"
type_synonym ('a, 's, 'tview) JointView = "'a ⇒ 's Trace ⇒ 'tview"
```

We require views to be *synchronous*, i.e. that agents be able to tell the time using their view by distinguishing two traces of different lengths. As we will see in the next section, this guarantees that the JKBP has an essentially unique implementation.

We extend the PreEnvironment locale with a synchronous view:

```
locale PreEnvironmentJView =
  PreEnvironment jkbp envInit envAction envTrans envVal
    for jkbp :: "('a, 'p, 'aAct) JKBP"
    and envInit :: "'s list"
    and envAction :: "'s ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's"
    and envVal :: "'s ⇒ 'p ⇒ bool"
+ fixes jview :: "('a, 's, 'tview) JointView"
  assumes sync: "∀a t t'. jview a t = jview a t' ⟶ tLength t = tLength t'"
```

The two principal synchronous views are the clock view and the perfect-recall view, both of which we discuss further in §3.7. We will derive an agent's concrete view from its instantaneous observations of the global state in §3.6.1.

We build an $S5_n$ structure from a set of traces by relating traces that yield the same view, and by evaluating propositions on the final state of a trace.

```
definition (in PreEnvironmentJView)
```

```
  mkM :: "'s Trace set ⇒ ('a, 'p, 's Trace) KripkeStructure"
where "mkM T ≡ ⦇ worlds = T,
                relations = λa. {(t, t'). {t, t'} ⊆ T ∧ jview a t = jview a t'},
                valuation = envVal ∘ tLast ⦈"
```

We now show how to generate a set of traces for a JKBP in an environment for a given view.

3.5 Canonical structures

We inductively define an *interpretation* of a JKBP with respect to an arbitrary set of traces T by constructing a sequence of sets of traces of increasing length:

```
fun jkbpTn :: "nat ⇒ 's Trace set ⇒ 's Trace set" where
  "jkbpT_0 T     = { tInit s |s. s ∈ set envInit }"
| "jkbpT_Suc n T = { t ↝ envTrans eact aact (tLast t) |t eact aact.
                          t ∈ jkbpT_n T ∧ eact ∈ set (envAction (tLast t))
                        ∧ (∀a. aact a ∈ set (jAction (mkM T) t a)) }"
```

The union of this sequence gives us a closure property:

```
definition jkbpT :: "'s Trace set ⇒ 's Trace set" where "jkbpT T ≡ ⋃n. jkbpT_n T"
```

We say that a set of traces T *represents* a JKBP if it is closed under jkbpT:

```
definition represents :: "'s Trace set ⇒ bool" where "represents T ≡ jkbpT T = T"
```

We break this vicious cycle using our assumption that the view is synchronous. Specifically, the actions of the JKBP on a trace t are a function of the traces in the model with the same length:

```
lemma sync_jview_jAction_eq:
  assumes traces: "{ t ∈ T . tLength t = n } = { t ∈ T' . tLength t = n }"
  assumes tT: "t ∈ { t ∈ T . tLength t = n }"
  shows "jAction (mkM T) t = jAction (mkM T') t"
```

This implies that for a synchronous view we can inductively define the *canonical traces* of a JKBP. These are the traces that a JKBP generates when it is interpreted with respect to those very same traces. We do this by constructing the sequence jkbpC$_n$ of *(canonical) temporal slices* similarly to jkbpT$_n$, and also its limit jkbpC and the corresponding $S5_n$ structures:

```
fun jkbpCn :: "nat ⇒ 's Trace set" where
  "jkbpC_0       = { tInit s |s. s ∈ set envInit }"
| "jkbpC_Suc n = { t ↝ envTrans eact aact (tLast t) |t eact aact.
                          t ∈ jkbpC_n ∧ eact ∈ set (envAction (tLast t))
                        ∧ (∀a. aact a ∈ set (jAction (mkM jkbpC_n) t a)) }"

abbreviation MCn :: "nat ⇒ ('a, 'p, 's Trace) KripkeStructure" where
  "MC_n ≡ mkM jkbpC_n"
```

```
definition jkbpC :: "'s Trace set" where
  "jkbpC ≡ ⋃n. jkbpCn"

abbreviation MC :: "('a, 'p, 's Trace) KripkeStructure" where
  "MC ≡ mkM jkbpC"
```

We can show that jkbpC represents the joint knowledge-based program `jkbp`:

```
lemma jkbpC_jkbpCn_jAction_eq:
  assumes "t ∈ jkbpCn"
  shows "jAction MC t = jAction MCn t"

lemma jkbpTn_jkbpCn_represents: "jkbpTn jkbpC = jkbpCn"
  by (induct n) (fastforce simp: Let_def jkbpC_jkbpCn_jAction_eq)+

theorem jkbpC_represents: "represents jkbpC"
```

We can show uniqueness too, by a similar argument:

```
theorem jkbpC_represents_uniquely:
  assumes "represents T"
  shows "T = jkbpC"
end
```

Thus, at least with synchronous views, we are justified in talking about *the* representation of a JKBP in a given environment. These results are also valid for the more general notion of *provides witnesses* as shown by Fagin et al. (1995, Lemma 7.2.4) and Fagin et al. (1997): it requires only that if a subjective knowledge formula is false on a trace then there is a trace of the same length or less that bears witness to that effect. This is useful in asynchronous settings.

The next section shows how we can construct canonical representations of JKBPs using automata.

3.6 Automata construction

Our attention now shifts to the question of how we can construct standard automata that *implement* a JKBP. We proceed by defining *incremental views* following van der Meyden (1996b), which provide the interface between the system and these automata. The algorithm itself is presented in §3.6.7.

3.6.1 Incremental views

Intuitively an agent instantaneously observes the system state, and so must maintain its view of the system *incrementally*: the new view must be a function of the current view and some new observation. We allow these observations to be an arbitrary projection of the system state:

```
locale Environment =
  PreEnvironment jkbp envInit envAction envTrans envVal
    for jkbp :: "('a, 'p, 'aAct) JKBP"
    and envInit :: "'s list"
    and envAction :: "'s ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's"
    and envVal :: "'s ⇒ 'p ⇒ bool"
+ fixes envObs :: "'a ⇒ 's ⇒ 'obs"
```

An incremental view therefore consists of two functions with these types:

```
type_synonym ('a, 'obs, 'tv) InitialIncrJointView = "'a ⇒ 'obs ⇒ 'tv"
type_synonym ('a, 'obs,  'tv) IncrJointView = "'a ⇒ 'obs ⇒ 'tv ⇒ 'tv"
```

These functions are required to commute with their corresponding trace-based joint view:

```
locale IncrEnvironment =
  Environment jkbp envInit envAction envTrans envVal envObs
+ PreEnvironmentJView jkbp envInit envAction envTrans envVal jview
    for jkbp :: "('a, 'p, 'aAct) JKBP"
    and envInit :: "'s list"
    and envAction :: "'s ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's"
    and envVal :: "'s ⇒ 'p ⇒ bool"
    and jview :: "('a, 's, 'tv) JointView"
    and envObs :: "'a ⇒ 's ⇒ 'obs"
+ fixes jviewInit :: "('a, 'obs, 'tv) InitialIncrJointView"
  fixes jviewIncr :: "('a, 'obs, 'tv) IncrJointView"
  assumes jviewInit: "∀a s. jviewInit a (envObs a s) = jview a (tInit s)"
  assumes jviewIncr: "∀a t s. jview a (t ⇝ s)
                             = jviewIncr a (envObs a s) (jview a t)"
```

Armed with these definitions we now show that there are automata that implement a JKBP in a given environment with respect to an arbitrary incremental synchronous view.

3.6.2 Automata and the notion of implementation

Our implementations of JKBPs take the form of deterministic Moore automata, where transitions are labelled by observations and states with the actions to be performed. We will use the term *protocols* interchangeably with automata, following the KBP literature, and adopt *joint protocols* for the assignment of one such to each agent:

```
record ('obs, 'aAct, 'ps) Protocol =
  pInit :: "'obs ⇒ 'ps"
  pTrans :: "'obs ⇒ 'ps ⇒ 'ps"
  pAct :: "'ps ⇒ 'aAct list"
```

```
type_synonym ('a, 'obs, 'aAct, 'ps) JointProtocol
    = "'a ⇒ ('obs, 'aAct, 'ps) Protocol"

context IncrEnvironment
begin
```

To ease composition with the system we adopt the function pInit which maps the initial observation to an initial automaton state. Intuitively all uncertainty the agent has about the system is already encoded into each automaton state, and so deterministic transitions are sufficient. In contrast we model the non-deterministic choice of action by making pAct set-valued.

Running a joint protocol on a trace is entirely standard, as is determining the agents' actions:

```
fun runJP :: "('a, 'obs, 'aAct, 'ps) JointProtocol ⇒ 's Trace ⇒ 'a ⇒ 'ps"
where
  "runJP jp (tInit s) a = pInit (jp a) (envObs a s)"
| "runJP jp (t ⇝ s) a = pTrans (jp a) (envObs a s) (runJP jp t a)"

abbreviation actJP :: "('a, 'obs, 'aAct, 'ps) JointProtocol
                      ⇒ 's Trace ⇒ 'a ⇒ 'aAct list"
where "actJP jp ≡ λt a. pAct (jp a) (runJP jp t a)"
```

Similarly to §3.5 we define the set of traces generated by a joint protocol in a fixed environment:

```
inductive_set
  jpTraces :: "('a, 'obs, 'aAct, 'ps) JointProtocol ⇒ 's Trace set"
    for jp :: "('a, 'obs, 'aAct, 'ps) JointProtocol"
where
  "s ∈ set envInit ⟹ tInit s ∈ jpTraces jp"
| "⟦ t ∈ jpTraces jp; eact ∈ set (envAction (tLast t));
     ⋀a. aact a ∈ set (actJP jp t a); s = envTrans eact aact (tLast t) ⟧
     ⟹ t ⇝ s ∈ jpTraces jp"

end
```

With this machinery in hand, we now relate automata with JKBPs. We say a joint protocol jp *implements* a JKBP when they perform the same actions on the canonical traces. Note that the behaviour of jp on other traces is arbitrary.

```
context IncrEnvironment
begin

definition implements :: "('a, 'obs, 'aAct, 'ps) JointProtocol ⇒ bool"
where "implements jp ≡ (∀t ∈ jkbpC. set ∘ actJP jp t = set ∘ jAction MC t)"
```

Clearly there are environments where the canonical trace set jkbpC can be generated by actions that differ from those prescribed by the JKBP. We can show that the *implements* relation is a stronger requirement than the mere trace-inclusion required by the *represents* relation of §3.5.

```
lemma implements_represents:
  assumes "implements jp"
  shows "represents (jpTraces jp)"
```

The proof is by a straightfoward induction over the lengths of traces generated by the joint protocol.

Our final piece of technical machinery allows us to refine automata definitions: we say that two joint protocols are *behaviourally equivalent* if the actions they propose coincide for each canonical trace. The implementation relation is preserved by this relation.

```
definition behaviourally_equiv :: "('a, 'obs, 'aAct, 'ps) JointProtocol
                                    ⇒ ('a, 'obs, 'aAct, 'ps') JointProtocol ⇒ bool"
where
  "behaviourally_equiv jp jp' ≡ ∀t ∈ jkbpC. set ∘ actJP jp t = set ∘ actJP jp' t"

lemma behaviourally_equiv_implements:
  assumes "behaviourally_equiv jp jp'"
  shows "implements jp ⟷ implements jp'"

end
```

3.6.3 Automata using equivalence classes

We now show that there is an implementation of every JKBP with respect to every incremental synchronous view. Intuitively the states of the automaton for agent `a` represent the equivalence classes of traces that `a` considers possible, and the transitions update these sets according to the JKBP and new observation.

```
context IncrEnvironment
begin

definition mkAutoEC :: "('a, 'obs, 'aAct, 's Trace set) JointProtocol"
where "mkAutoEC ≡ λa. ⦇ pInit = λobs. { t ∈ jkbpC . jviewInit a obs = jview a t },
                        pTrans = λobs ps. { t |t t'. t ∈ jkbpC ∧ t' ∈ ps
                                      ∧ jview a t = jviewIncr a obs (jview a t') },
                        pAct = λps. jAction MC (SOME t. t ∈ ps) a ⦈"
```

The function SOME is Hilbert's indefinite description operator ε, used here to choose an arbitrary trace from the protocol state.

That this automaton maintains the correct equivalence class on a trace `t` follows from an easy induction over `t`.

```
lemma mkAutoEC_ec:
  assumes "t ∈ jkbpC"
  shows "runJP mkAutoEC t a = { t' ∈ jkbpC . jview a t' = jview a t }"
```

We can show that the construction yields an implementation by appealing to the previous lemma and showing that the pAct functions coincide.

```
lemma mkAutoEC_implements: "implements mkAutoEC"
```

This definition leans on the canonical trace set jkbpC, and is indeed effective: we can enumerate the canonical traces and are sure to find one that has the view we expect. Then it is sufficient to consider other traces of the same length due to synchrony. We would need to do this computation dynamically, as the automaton will (in general) have an infinite state space.

end

3.6.4 Automata using simulations

Our goal now is to reduce the space required by the automaton constructed by mkAutoEC by *simulating* the equivalence classes (§3.2.3).

The following locale captures the framework of van der Meyden (1996b):

```
locale SimIncrEnvironment =
  IncrEnvironment jkbp envInit envAction envTrans envVal jview envObs
                  jviewInit jviewIncr
    for jkbp :: "('a, 'p, 'aAct) JKBP"

    and envInit :: "'s list"
    and envAction :: "'s ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's"
    and envVal :: "'s ⇒ 'p ⇒ bool"
    and jview :: "('a, 's, 'tv) JointView"
    and envObs :: "'a ⇒ 's ⇒ 'obs"
    and jviewInit :: "('a, 'obs, 'tv) InitialIncrJointView"
    and jviewIncr :: "('a, 'obs, 'tv) IncrJointView"
+ fixes simf :: "'s Trace ⇒ 'ss"
  fixes simRels :: "'a ⇒ ('ss × 'ss) set"
  fixes simVal :: "'ss ⇒ 'p ⇒ bool"
  assumes simf: "sim MC (mkKripke (simf ` jkbpC) simRels simVal) simf"

context SimIncrEnvironment
begin
```

Note that the back tick ` is Isabelle/HOL's relational image operator. In context it says that simf must be a simulation from jkbpC to its image under simf.

Firstly we lift our canonical trace sets and Kripke structures through the simulation.

```
abbreviation jkbpCSn :: "nat ⇒ 'ss set" where "jkbpCSn ≡ simf ` jkbpCn"
abbreviation jkbpCS :: "'ss set" where "jkbpCS ≡ simf ` jkbpC"

abbreviation MCSn :: "nat ⇒ ('a, 'p, 'ss) KripkeStructure" where
```

```
  "MCSn ≡ mkKripke jkbpCSn simRels simVal"

abbreviation MCS :: "('a, 'p, 'ss) KripkeStructure" where
  "MCS ≡ mkKripke jkbpCS simRels simVal"
```

We often use the equivalence class of simulated traces generated by agent a's view:

```
abbreviation sim_equiv_class :: "'a ⇒ 's Trace ⇒ 'ss set" where
  "sim_equiv_class a t ≡ simf ` { t' ∈ jkbpC . jview a t' = jview a t }"

abbreviation jkbpSEC :: "'ss set set" where
  "jkbpSEC ≡ ⋃a. sim_equiv_class a ` jkbpC"
```

We can show that the temporal slice of the simulated structure is adequate for determining the actions of the JKBP. The proof is routine but tedious, exploiting the sub-model property (§3.2.2).

```
lemma jkbpC_jkbpCSn_jAction_eq:
  assumes tCn: "t ∈ jkbpCn n"
  shows "jAction MC t = jAction (MCSn n) (simf t)"
end
```

It can be shown that a suitable simulation into a finite structure is adequate to establish the existence of finite-state implementations (van der Meyden 1996b, Theorem 2): essentially we apply the simulation to the states of mkAutoEC. However this result does not make it clear how the transition function can be algorithmically constructed. One approach is to maintain jkbpC while extending the automaton, which is quite space inefficient.

Intuitively we wish to compute the possible sim_equiv_class successors of a given sim_equiv_class without reference to jkbpC, and this should be possible as the reachable simulated worlds must contain enough information to differentiate themselves from every other simulated world that represents a trace on which the agents act differently.

This leads us to ask for some extra functionality of our simulation, which we detail in the locale shown in Figure 3.1. Note that these definitions are stated relative to the environment and the JKBP, allowing us to treat specialised cases such as having a single agent (§3.7.3) and broadcast environments (§3.7.4 and §3.7.5).

Firstly we relate the concrete representation 'rep of equivalence classes under simulation to differ from the abstract representation 'ss set using the abstraction function simAbs; there is no one-size-fits-all concrete representation, as we will see.

Secondly we ask for a function simInit a iobs that faithfully generates a representation of the equivalence class of simulated initial states that are possible for agent a given the valid initial observation iobs.

Thirdly the simObs function allows us to partition the results of simTrans according to the recurrent observation that agent a makes of the equivalence class.

```
locale AlgSimIncrEnvironment =
  SimIncrEnvironment jkbp envInit envAction envTrans envVal
                     jview envObs jviewInit jviewIncr simf simRels simVal
    for jkbp :: "('a, 'p, 'aAct) JKBP"
    and envInit :: "'s list"
    and envAction :: "'s ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's"
    and envVal :: "'s ⇒ 'p ⇒ bool"

    and jview :: "('a, 's, 'tv) JointView"
    and envObs :: "'a ⇒ 's ⇒ 'obs"
    and jviewInit :: "('a, 'obs, 'tv) InitialIncrJointView"
    and jviewIncr :: "('a, 'obs, 'tv) IncrJointView"

    and simf :: "'s Trace ⇒ 'ss"
    and simRels :: "'a ⇒ ('ss × 'ss) set"
    and simVal :: "'ss ⇒ 'p ⇒ bool"

+ fixes simAbs :: "'rep ⇒ 'ss set"

    and simObs :: "'a ⇒ 'rep ⇒ 'obs"
    and simInit :: "'a ⇒ 'obs ⇒ 'rep"
    and simTrans :: "'a ⇒ 'rep ⇒ 'rep list"
    and simAction :: "'a ⇒ 'rep ⇒ 'aAct list"

  assumes simInit:
          "∀a iobs. iobs ∈ envObs a ‘ set envInit
                 ⟶ simAbs (simInit a iobs)
                   = simf ‘ { t' ∈ jkbpC. jview a t' = jviewInit a iobs }"
      and simObs:
          "∀a ec t. t ∈ jkbpC ∧ simAbs ec = sim_equiv_class a t
                 ⟶ simObs a ec = envObs a (tLast t)"
      and simAction:
          "∀a ec t. t ∈ jkbpC ∧ simAbs ec = sim_equiv_class a t
                 ⟶ set (simAction a ec) = set (jAction MC t a)"
      and simTrans:
          "∀a ec t. t ∈ jkbpC ∧ simAbs ec = sim_equiv_class a t
                 ⟶ simAbs ‘ set (simTrans a ec)
                   = { sim_equiv_class a (t' ⇝ s)
                       |t' s. t' ⇝ s ∈ jkbpC ∧ jview a t' = jview a t }"
```

Figure 3.1: The SimEnvironment locale extends the Environment locale with simulation and algorithmic operations. The backtick ‘ is Isabelle/HOL's image-of-a-set-under-a-function operator.

Fourthly, the function `simAction` computes a list of actions enabled by the JKBP on a state that concretely represents a canonical equivalence class.

Finally we expect to compute the list of represented sim_equiv_class successors of a given sim_equiv_class using `simTrans`.

With these functions in hand, we can define our desired automaton:

```
definition (in AlgSimIncrEnvironment)
      mkAutoSim :: "('a, 'obs, 'aAct, 'rep) JointProtocol"
where "mkAutoSim ≡ λa.
  ⦇ pInit = simInit a,
    pTrans = λobs ec. (SOME ec'. ec' ∈ set (simTrans a ec) ∧ simObs a ec' = obs),
    pAct = simAction a ⦈"
```

The automaton faithfully constructs the simulated equivalence class of the given trace:

```
lemma (in AlgSimIncrEnvironment) mkAutoSim_ec:
  assumes "t ∈ jkbpC"
  shows "simAbs (runJP mkAutoSim t a) = sim_equiv_class a t"
```

It is then a short step to the following version of Theorem 2 of van der Meyden (1996b):

```
theorem (in AlgSimIncrEnvironment) mkAutoSim_implements: "implements mkAutoSim"
```

The reader may care to contrast these structures with the *progression structures* of van der Meyden (1996c), where states contain entire Kripke structures, and expanding the automaton is alternated with bisimulation reduction to ensure termination when a finite-state implementation exists (see §6.2.4) We also use simulations in Appendix A to show the complexity of some related model checking problems.

We now review a simple *depth-first search* (DFS) theory, and an abstraction of finite maps, before presenting the algorithm for constructing implementations of KBPs.

3.6.5 Generic DFS

We use a generic DFS to construct the transitions and action function of the implementation of the JKBP, though any complete traversal strategy of the state space would suffice for correctness. This theory is an adaptation of the work of S. Berghofer and A. Krauss (see Berghofer and Reiter (2009)) to our data refinement setting.

The DFS itself is defined in the standard tail-recursive way:

```
partial_function (tailrec) gen_dfs where
  "gen_dfs succs ins memb S wl = (case wl of
     [] ⇒ S
   | (x # xs) ⇒
       if memb x S then gen_dfs succs ins memb S xs
       else gen_dfs succs ins memb (ins x S) (succs x @ xs))"
```

```
locale DFS =
  fixes succs :: "'a ⇒ 'a list"
  and isNode :: "'a ⇒ bool"
  and invariant :: "'b ⇒ bool"
  and ins :: "'a ⇒ 'b ⇒ 'b"
  and memb :: "'a ⇒ 'b ⇒ bool"
  and empt :: 'b
  and nodeAbs :: "'a ⇒ 'c"
  assumes ins_eq: "⋀x y S. ⟦ isNode x; isNode y; invariant S; ¬ memb y S ⟧
                      ⟹ memb x (ins y S)
                      ⟷ ((nodeAbs x = nodeAbs y) ∨ memb x S)"
  and succs: "⋀x y. ⟦ isNode x; isNode y; nodeAbs x = nodeAbs y ⟧
                      ⟹ nodeAbs ‘ set (succs x) = nodeAbs ‘ set (succs y)"
  and empt: "⋀x. isNode x ⟹ ¬ memb x empt"
  and succs_isNode: "⋀x. isNode x ⟹ list_all isNode (succs x)"
  and empt_invariant: "invariant empt"
  and ins_invariant: "⋀x S. ⟦ isNode x; invariant S; ¬ memb x S ⟧
                       ⟹ invariant (ins x S)"
  and graph_finite: "finite (nodeAbs ‘ { x . isNode x})"
```

Figure 3.2: The DFS locale.

The proofs are carried out in the locale of Figure 3.2, which details our requirements on the parameters for the DFS to behave as one would expect. Intuitively we are traversing a graph defined by `succs` from some initial work list `wl`, constructing an object of type `'b` as we go. The function `ins` integrates the current node into this construction. The predicate `isNode` is invariant over the set of states reachable from the initial work list, and is respected by `empt` and `ins`. We can also supply an invariant for the constructed object (`invariant`). Inside the locale, `dfs` abbreviates `gen_dfs` partially applied to the fixed parameters.

To support our data refinement (§3.6.4) we also require that the representation of nodes be adequate via the abstraction function `nodeAbs`, which the transition relation `succs` and visited predicate `memb` must respect. To ensure termination it must be the case that there are only a finite number of states, though there might be an infinity of representations.

We characterise the DFS traversal using the reflexive transitive closure operator:

```
definition (in DFS) reachable :: "'a set ⇒ 'a set" where
  "reachable xs ≡ {(x, y). y ∈ set (succs x)}* ‘‘ xs"
```

We make use of two results about the traversal. Firstly, some representation of each reachable node is incorporated into the final construction:

```
theorem (in DFS) reachable_imp_dfs:
  assumes y: "isNode y"
      and xs: "list_all isNode xs"
      and m: "y ∈ reachable (set xs)"
  shows "∃y'. nodeAbs y' = nodeAbs y ∧ memb y' (dfs empt xs)"
```

Secondly, that if an invariant holds on the initial object then it holds on the final one:

```
theorem (in DFS) dfs_invariant:
  assumes "invariant S"
  assumes "list_all isNode xs"
  shows "invariant (dfs S xs)"
```

3.6.6 Finite map operations

The algorithm represents automata as pairs of finite maps, which we capture as follows:

```
record ('m, 'k, 'e) MapOps =
  empty :: "'m"
  lookup :: "'m ⇒ 'k ⇀ 'e"
  update :: "'k ⇒ 'e ⇒ 'm ⇒ 'm"

definition MapOps :: "('k ⇒ 'kabs) ⇒ 'kabs set ⇒ ('m, 'k, 'e) MapOps ⇒ bool"
where "MapOps α d ops ≡ (∀k. α k ∈ d ⟶ lookup ops (empty ops) k = None)
                        ∧ (∀e k k' M. α k ∈ d ∧ α k' ∈ d
                             ⟶ lookup ops (update ops k e M) k'
                             = (if α k' = α k then Some e else lookup ops M k'))"
```

The function α abstracts concrete keys of type `'k`, and the parameter `d` specifies the valid abstract keys. This approach has the advantage over a locale that we can pass records to functions, while for a locale we would need to pass the three functions separately (as in the DFS theory of §3.6.5) as Isabelle's code generator presently does not understand locales.

We use the following function to test for membership in the domain of the map:

```
definition isSome :: "'a option ⇒ bool" where
  "isSome opt ≡ case opt of None ⇒ False | Some _ ⇒ True"
```

3.6.7 An algorithm for automata construction

We now construct the automaton defined by mkAutoSim (§3.6.4) using the DFS of §3.6.5. From here on we assume that the environment consists of only a finite set of states, using the FiniteEnvironment locale shown in Figure 3.3.

The Algorithm locale, also shown in Figure 3.3, extends the AlgSimIncrEnvironment locale with a pair of finite map operations: `aOps` is used to map automata states to lists of actions, and `tOps` handles simulated transitions. In both cases the maps are only required to work on the abstract domain of simulated canonical traces. Note also that the space of simulated equivalence classes of type `'ss` must be finite but there is no restriction on the representation type `'rep`.

We develop the algorithm for a single, fixed agent, which requires us to define a new locale AlgorithmForAgent that extends Algorithm with an extra parameter designating the agent:

```
locale AlgorithmForAgent =
  Algorithm jkbp envInit envAction envTrans envVal jview envObs jviewInit jviewIncr
            simf simRels simVal simAbs simObs simInit simTrans simAction
```

```
locale FiniteEnvironment =
  Environment jkbp envInit envAction envTrans envVal envObs
    for jkbp :: "('a, 'p, 'aAct) JKBP"
    and envInit :: "('s :: finite) list"
    and envAction :: "'s ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's"
    and envVal :: "'s ⇒ 'p ⇒ bool"
    and envObs :: "'a ⇒ 's ⇒ 'obs"

locale Algorithm =
  FiniteEnvironment jkbp envInit envAction envTrans envVal envObs
+ AlgSimIncrEnvironment jkbp envInit envAction envTrans envVal jview envObs
                        jviewInit jviewIncr simf simRels simVal simAbs simObs
                        simInit simTrans simAction
    for jkbp :: "('a, 'p, 'aAct) JKBP"
    and envInit :: "('s :: finite) list"
    and envAction :: "'s ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's"
    and envVal :: "'s ⇒ 'p ⇒ bool"
    and jview :: "('a, 's, 'tobs) JointView"

    and envObs :: "'a ⇒ 's ⇒ 'obs"
    and jviewInit :: "('a, 'obs, 'tobs) InitialIncrJointView"
    and jviewIncr :: "('a, 'obs, 'tobs) IncrJointView"

    and simf :: "'s Trace ⇒ 'ss :: finite"
    and simRels :: "'a ⇒ ('ss × 'ss) set"
    and simVal :: "'ss ⇒ 'p ⇒ bool"

    and simAbs :: "'rep ⇒ 'ss set"

    and simObs :: "'a ⇒ 'rep ⇒ 'obs"
    and simInit :: "'a ⇒ 'obs ⇒ 'rep"
    and simTrans :: "'a ⇒ 'rep ⇒ 'rep list"
    and simAction :: "'a ⇒ 'rep ⇒ 'aAct list"

+ fixes aOps :: "('ma, 'rep, 'aAct list) MapOps"
    and tOps :: "('mt, 'rep × 'obs, 'rep) MapOps"

  assumes aOps: "MapOps simAbs jkbpSEC aOps"
      and tOps: "MapOps (λk. (simAbs (fst k), snd k)) (jkbpSEC × UNIV) tOps"
```

Figure 3.3: The FiniteEnvironment and Algorithm locales.

```
          aOps tOps
  — ...
+ fixes a :: "'a"
```

DFS operations

We represent the automaton under construction using a record:

```
record ('ma, 'mt) AlgState =
  aActs :: "'ma"
  aTrans :: "'mt"

context AlgorithmForAgent
begin
```

We instantiate the DFS theory with the following functions.

A node is an equivalence class of represented simulated traces.

```
definition k_isNode :: "'rep ⇒ bool" where
  "k_isNode ≡ λec. simAbs ec ∈ sim_equiv_class a ` jkbpC"
```

The successors of a node are those produced by the simulated transition function.

```
abbreviation k_succs :: "'rep ⇒ 'rep list" where
  "k_succs ≡ simTrans a"
```

The initial automaton has no transitions and no actions.

```
definition k_empt :: "('ma, 'mt) AlgState" where
  "k_empt ≡ ⦇ aActs = empty aOps, aTrans = empty tOps ⦈"
```

The domain of the action map tracks the set of nodes the DFS has visited.

```
definition k_memb :: "'rep ⇒ ('ma, 'mt) AlgState ⇒ bool" where
  "k_memb s A ≡ isSome (lookup aOps (aActs A) s)"
```

We add a new equivalence class to the automaton by updating the action and transition maps.

```
definition actsUpdate :: "'rep ⇒ ('ma, 'mt) AlgState ⇒ 'ma" where
  "actsUpdate ec A ≡ update aOps ec (simAction a ec) (aActs A)"

definition transUpdate :: "'rep ⇒ 'rep ⇒ 'mt ⇒ 'mt" where
  "transUpdate ec ec' at ≡ update tOps (ec, simObs a ec') ec' at"

definition k_ins :: "'rep ⇒ ('ma, 'mt) AlgState ⇒ ('ma, 'mt) AlgState" where
  "k_ins ec A ≡ ⦇ aActs = actsUpdate ec A,
                  aTrans = foldr (transUpdate ec) (k_succs ec) (aTrans A) ⦈"
```

The required properties are straightforward to show.

Algorithm invariant

At each step of the process the state represents an automaton that concords with mkAutoSim on the visited equivalence classes. We also need to know that the state has preserved the MapOps invariants.

```
definition k_invariant :: "('ma, 'mt) AlgState ⇒ bool" where
  "k_invariant A ≡
      (∀ec ec'. k_isNode ec ∧ k_isNode ec' ∧ simAbs ec' = simAbs ec
        ⟶ lookup aOps (aActs A) ec = lookup aOps (aActs A) ec')
    ∧ (∀ec ec' obs. k_isNode ec ∧ k_isNode ec' ∧ simAbs ec' = simAbs ec
        ⟶ lookup tOps (aTrans A) (ec, obs) = lookup tOps (aTrans A) (ec', obs))
    ∧ (∀ec. k_isNode ec ∧ k_memb ec A
        ⟶ (∃acts. lookup aOps (aActs A) ec = Some acts
                 ∧ set acts = set (simAction a ec)))
    ∧ (∀ec obs. k_isNode ec ∧ k_memb ec A
              ∧ obs ∈ simObs a ` set (simTrans a ec)
        ⟶ (∃ec'. lookup tOps (aTrans A) (ec, obs) = Some ec'
                ∧ simAbs ec' ∈ simAbs ` set (simTrans a ec)
                ∧ simObs a ec' = obs))"
```

Showing that the invariant holds of k_empt and is respected by k_ins is routine.

The initial frontier is the partition of the set of initial states under the initial observation function.

```
definition (in Algorithm) k_frontier :: "'a ⇒ 'rep list" where
  "k_frontier a ≡ map (simInit a ∘ envObs a) envInit"
```

We now instantiate the DFS locale with respect to the AlgorithmForAgent locale. The instantiated lemmas are given the mandatory prefix KBPAlg in the AlgorithmForAgent locale.

```
sublocale AlgorithmForAgent
        < KBPAlg!: DFS k_succs k_isNode k_invariant k_ins k_memb k_empt simAbs
```

The final algorithm, with the constants inlined, is shown in Figure 3.4. The rest of this section shows its correctness.

It follows immediately from dfs_invariant that the invariant holds of the result of the DFS:

```
lemma k_dfs_invariant: "k_invariant k_dfs"
```

The set of reachable equivalence classes coincides with the partition of jkbpC under the simulation and representation functions:

```
lemma k_reachable:
  "simAbs ` KBPAlg.reachable (set (k_frontier a)) = sim_equiv_class a ` jkbpC"
```

Left to right follows from an induction on the reflexive, transitive closure, and right to left by induction over canonical traces.

This result immediately yields the same result at the level of representations:

```
definition
  alg_dfs :: "('ma, 'rep, 'aAct list) MapOps
         ⇒ ('mt, 'rep × 'obs, 'rep) MapOps
         ⇒ ('rep ⇒ 'obs)
         ⇒ ('rep ⇒ 'rep list)
         ⇒ ('rep ⇒ 'aAct list)
         ⇒ 'rep list
         ⇒ ('ma, 'mt) AlgState"
where
  "alg_dfs aOps tOps simObs simTrans simAction ≡
    let k_empt = ⦇ aActs = empty aOps, aTrans = empty tOps ⦈;
       k_memb = (λs A. isSome (lookup aOps (aActs A) s));
       k_succs = simTrans;
       actsUpdate = λec A. update aOps ec (simAction ec) (aActs A);
       transUpdate = λec ec' at. update tOps (ec, simObs ec') ec' at;
       k_ins = λec A. ⦇ aActs = actsUpdate ec A,
                     aTrans = foldr (transUpdate ec) (k_succs ec) (aTrans A) ⦈
     in gen_dfs k_succs k_ins k_memb k_empt"

definition
  mkAlgAuto :: "('ma, 'rep, 'aAct list) MapOps
           ⇒ ('mt, 'rep × 'obs, 'rep) MapOps
           ⇒ ('a ⇒ 'rep ⇒ 'obs)
           ⇒ ('a ⇒ 'obs ⇒ 'rep)
           ⇒ ('a ⇒ 'rep ⇒ 'rep list)
           ⇒ ('a ⇒ 'rep ⇒ 'aAct list)
           ⇒ ('a ⇒ 'rep list)
           ⇒ ('a, 'obs, 'aAct, 'rep) JointProtocol"
where
  "mkAlgAuto aOps tOps simObs simInit simTrans simAction frontier ≡ λa.
    let auto = alg_dfs aOps tOps (simObs a) (simTrans a) (simAction a)
                   (frontier a)
     in ⦇ pInit = simInit a,
          pTrans = λobs ec. the (lookup tOps (aTrans auto) (ec, obs)),
          pAct = λec. the (lookup aOps (aActs auto) ec) ⦈"
```

Figure 3.4: The algorithm. The function the projects a value from the 'a option type.

```
lemma k_memb_rep:
  assumes "k_isNode rec"
  shows "k_memb rec k_dfs"
end
```

This concludes our agent-specific reasoning; we now show that the algorithm works for all agents. The following command generalises all our lemmas in the AlgorithmForAgent to the Algorithm locale, giving them the mandatory prefix KBP:

```
sublocale Algorithm
  < KBP!: AlgorithmForAgent jkbp envInit envAction envTrans envVal jview envObs
                            jviewInit jviewIncr simf simRels simVal simAbs simObs
                            simInit simTrans simAction aOps tOps a for a
context Algorithm
begin

abbreviation "k_mkAlgAuto ≡
  mkAlgAuto aOps tOps simObs simInit simTrans simAction k_frontier"
```

Running the automata produced by the DFS on a canonical trace t yields some representation of the expected equivalence class:

```
lemma k_mkAlgAuto_ec:
  assumes "t ∈ jkbpC"
  shows "simAbs (runJP k_mkAlgAuto t a) = sim_equiv_class a t"
```

That the DFS and mkAutoSim yield the same actions on canonical traces follows immediately from this result and the invariant:

```
lemma k_mkAlgAuto_mkAutoSim_act_eq:
  assumes "t ∈ jkbpC"
  shows "set ∘ actJP k_mkAlgAuto t = set ∘ actJP mkAutoSim t"
```

Therefore these two constructions are behaviourally equivalent, and so the DFS generates an implementation of jkbp in the given environment:

```
theorem k_mkAlgAuto_implements: "implements k_mkAlgAuto"
end
```

Clearly the automata generated by this algorithm are large. We discuss this issue in §6.2.4.

3.7 Concrete views

Following van der Meyden (1996b), we provide two concrete synchronous views that illustrate how the theory works. For each view we give a simulation and a representation that satisfy the requirements of the Algorithm locale in Figure 3.3.

3.7.1 The clock view

The *clock view* records the current time and the observation for the most recent state:

```
definition (in Environment) clock_jview :: "('a, 's, nat × 'obs) JointView" where
  "clock_jview ≡ λa t. (tLength t, envObs a (tLast t))"
```

This is the least-information synchronous view. We show that finite-state implementations exist for all environments with respect to this view as per van der Meyden (1996b).

The corresponding incremental view simply increments the counter and records the new observation.

```
definition (in Environment)
       clock_jviewInit :: "'a ⇒ 'obs ⇒ nat × 'obs"
where "clock_jviewInit ≡ λa obs. (0, obs)"

definition (in Environment)
       clock_jviewIncr :: "'a ⇒ 'obs ⇒ nat × 'obs ⇒ nat × 'obs"
where "clock_jviewIncr ≡ λa obs' (l, obs). (l + 1, obs')"
```

It is straightforward to demonstrate the assumptions of the incremental environment locale (§3.6.1) with respect to an arbitrary environment.

```
sublocale Environment
  < Clock!: IncrEnvironment jkbp envInit envAction envTrans envVal
                              clock_jview envObs clock_jviewInit clock_jviewIncr
```

As we later show, satisfaction of a formula at a trace $t \in$ Clock.jkbpC$_n$ is determined by the set of final states of traces in Clock.jkbpCn:

```
context Environment
begin

abbreviation clock_commonAbs :: "'s Trace ⇒ 's set" where
  "clock_commonAbs t ≡ tLast ` Clock.jkbpCn (tLength t)"
```

Intuitively this set contains the states that the agents commonly consider possible at time n, which is sufficient for determining knowledge as the clock view ignores paths. Therefore we can simulate trace t by pairing this abstraction of t with its final state:

```
type_synonym (in -) 's clock_simWorlds = "'s set × 's"

definition clock_sim :: "'s Trace ⇒ 's clock_simWorlds" where
  "clock_sim ≡ λt. (clock_commonAbs t, tLast t)"
```

In the Kripke structure for our simulation, we relate worlds for a if the sets of commonly-held states coincide, and the observation of the final states of the traces is the same. Propositions are evaluated at the final state.

```
definition clock_simRels :: "'a ⇒ ('s clock_simWorlds × 's clock_simWorlds) set" where
  "clock_simRels ≡ λa. { ((X, s), (X', s')) |X X' s s'.
                         X = X' ∧ {s, s'} ⊆ X ∧ envObs a s = envObs a s' }"

definition clock_simVal :: "'s clock_simWorlds ⇒ 'p ⇒ bool" where
  "clock_simVal ≡ envVal ∘ snd"

abbreviation clock_simMC :: "('a, 'p, 's clock_simWorlds) KripkeStructure" where
  "clock_simMC ≡ mkKripke (clock_sim ` Clock.jkbpC) clock_simRels clock_simVal"
```

That this is in fact a simulation (§3.2.3) is entirely straightforward.

```
lemma clock_sim: "sim Clock.MC clock_simMC clock_sim"
end
```

The SimIncrEnvironment of §3.6.4 only requires that we provide it an Environment and a simulation.

```
sublocale Environment
  < Clock!: SimIncrEnvironment jkbp envInit envAction envTrans envVal
                               clock_jview envObs clock_jviewInit clock_jviewIncr
                               clock_sim clock_simRels clock_simVal
```

We next consider algorithmic issues.

Representations

As the maps are keyed by equivalence classes of states, it is preferable that these sets have canonical representations. A simple approach is to use *ordered distinct lists* of type 'a odlist for the sets and *digital tries* (prefix trees) for the maps. Therefore we ask that environment states 's belong to the class linorder of linearly-ordered types, and moreover that the set agents be effectively presented. We introduce a new locale capturing these requirements:

```
locale FiniteLinorderEnvironment =
  Environment jkbp envInit envAction envTrans envVal envObs
    for jkbp :: "('a::{finite, linorder}, 'p, 'aAct) JKBP"
    and envInit :: "('s::{finite, linorder}) list"
    and envAction :: "'s ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's"
    and envVal :: "'s ⇒ 'p ⇒ bool"
    and envObs :: "'a ⇒ 's ⇒ 'obs"
+ fixes agents :: "'a odlist"
  assumes agents: "ODList.toSet agents = UNIV"

context FiniteLinorderEnvironment
begin
```

For a fixed agent a, we can reduce the number of worlds in clock_simMC by taking its quotient with respect to the equivalence relation for a. In other words, we represent a simulated equivalence class by pairing the set of all states reachable at that particular time with the subset of these that a considers possible. The worlds in our representational Kripke structure are therefore a pair of ordered, distinct lists:

```
type_synonym (in -) 's clock_simWorldsRep = "'s odlist × 's odlist"
```

We can readily abstract a representation to a set of simulated equivalence classes:

```
definition (in -)
  clock_simAbs :: "'s::linorder clock_simWorldsRep ⇒ 's clock_simWorlds set"
where
  "clock_simAbs X ≡ { (ODList.toSet (fst X), s) |s. s ∈ ODList.toSet (snd X) }"
```

Assuming X represents a simulated equivalence class for t ∈ jkbpC, clock_simAbs X decomposes into these two functions:

```
definition agent_abs :: "'a ⇒ 's Trace ⇒ 's set" where
  "agent_abs a t ≡
     { tLast t' |t'. t' ∈ Clock.jkbpC ∧ clock_jview a t' = clock_jview a t}"

definition common_abs :: "'s Trace ⇒ 's set" where
  "common_abs t ≡ tLast ` Clock.jkbpCn (tLength t)"
```

This representation is canonical on the domain of interest (though not in general):

```
lemma clock_simAbs_inj_on:
  "inj_on clock_simAbs { x . clock_simAbs x ∈ Clock.jkbpSEC }"
```

We could further compress this representation by labelling each element of the set of states reachable at time n with a bit to indicate whether the agent considers that state possible. This representation is, however, non-canonical: if (s, True) is in the representation, indicating that the agent considers s possible, then (s, False) may or may not be. The associated abstraction function is not injective and hence would obfuscate the following.

The following lemmas use a Kripke structure based on the set of final states of a temporal slice X:

```
definition clock_repRels :: "'a ⇒ ('s × 's) set" where
  "clock_repRels ≡ λa. { (s, s'). envObs a s = envObs a s' }"

abbreviation clock_repMC :: "'s set ⇒ ('a, 'p, 's) KripkeStructure" where
  "clock_repMC ≡ λX. mkKripke X clock_repRels envVal"
```

We show that this Kripke structure retains sufficient information from clock_simMC by exhibiting a simulation. This is eased by an intermediary structure that focuses on a particular trace:

```
abbreviation clock_jkbpCSt :: "'b Trace ⇒ 's clock_simWorlds set" where
  "clock_jkbpCSt t ≡ clock_sim ` Clock.jkbpCn (tLength t)"
```

```
abbreviation
        clock_simMCt :: "'b Trace ⇒ ('a, 'p, 's clock_simWorlds) KripkeStructure"
where "clock_simMCt t ≡ mkKripke (clock_jkbpCSt t) clock_simRels clock_simVal"

definition clock_repSim :: "'s clock_simWorlds ⇒ 's" where "clock_repSim ≡ snd"

lemma clock_repSim:
  "sim (clock_simMCt t) ((clock_repMC ∘ fst) (clock_sim t)) clock_repSim"
```

The following sections show how we satisfy the remaining requirements of the Algorithm locale of Figure 3.3. Where the proof is routine, we simply present the lemma without comment. The code generator in the present version of Isabelle (2012) can only handle top-level definitions, and not those inside a locale; we use the syntax (in -) to do this, and then define (but elide) locale-local abbreviations that supply the locale-bound variables to these definitions.

Initial states

An initial state of the automaton consists of envInit paired with the relevant equivalence class.

```
definition (in -) clock_simInit :: "('s::linorder) list ⇒ ('a ⇒ 's ⇒ 'obs)
                                  ⇒ 'a ⇒ 'obs ⇒ 's clock_simWorldsRep"
where "clock_simInit envInit envObs ≡ λa iobs.
  let cec = ODList.fromList envInit
   in (cec, ODList.filter (λs. envObs a s = iobs) cec)"

lemma clock_simInit:
  assumes "iobs ∈ envObs a ` set envInit"
  shows "clock_simAbs (clock_simInit a iobs)
       = clock_sim ` { t' ∈ Clock.jkbpC. clock_jview a t' = clock_jviewInit a iobs }"
```

Simulated observations

Agent a will make the same observation at any of the worlds that it considers possible, so we choose the first one in the list:

```
definition (in -) clock_simObs :: "('a ⇒ ('s :: linorder) ⇒ 'obs)
                                ⇒ 'a ⇒ 's clock_simWorldsRep ⇒ 'obs"
where "clock_simObs envObs ≡ λa. envObs a ∘ ODList.hd ∘ snd"

lemma clock_simObs:
  assumes "t ∈ Clock.jkbpC" and "clock_simAbs ec = Clock.sim_equiv_class a t"
  shows "clock_simObs a ec = envObs a (tLast t)"
```

Evaluation

We define our eval function in terms of evalS, which implements Boolean logic over 's odlist in the usual way – see §3.7.3 for the relevant clauses. It requires three functions specific to the representation: one each for propositions, knowledge and common knowledge.

Propositions define subsets of the worlds considered possible:

```
abbreviation (in -) clock_evalProp :: "(('s::linorder) ⇒ 'p ⇒ bool)
                                   ⇒ 's odlist ⇒ 'p ⇒ 's odlist"
where "clock_evalProp envVal ≡ λX p. ODList.filter (λs. envVal s p) X"
```

The knowledge relation computes the subset of the commonly-held-possible worlds cec that agent a considers possible at world s:

```
definition (in -) clock_knowledge :: "('a ⇒ ('s :: linorder) ⇒ 'obs) ⇒ 's odlist
                                  ⇒ 'a ⇒ 's ⇒ 's odlist"
where "clock_knowledge envObs cec ≡ λa s.
          ODList.filter (λs'. envObs a s = envObs a s') cec"
```

Similarly the common knowledge operation computes the transitive closure of the union of the knowledge relations for the agents as:

```
definition (in -) clock_commonKnowledge :: "('a ⇒ ('s :: linorder) ⇒ 'obs) ⇒ 's odlist
                                        ⇒ 'a list ⇒ 's ⇒ 's odlist"
where "clock_commonKnowledge envObs cec ≡ λas s.
     let r = λa. ODList.fromList [ (s', s'') . s' ← toList cec, s'' ← toList cec,
                                   envObs a s' = envObs a s'' ];
         R = toList (ODList.big_union r as)
      in ODList.fromList (memo_list_trancl R s)"
```

The function memo_list_trancl is from the executable transitive closure theory of Sternagel and Thiemann (2011).

The following function evaluates a subjective knowledge formula on the representation of an equivalence class:

```
definition (in -) eval :: "(('s :: linorder) ⇒ 'p ⇒ bool)
                        ⇒ ('a ⇒ 's ⇒ 'obs)
                        ⇒ 's clock_simWorldsRep ⇒ ('a, 'p) Kform ⇒ bool"
where "eval envVal envObs ≡ λ(cec, aec). evalS (clock_evalProp envVal)
                                            (clock_knowledge envObs cec)
                                            (clock_commonKnowledge envObs cec)
                                            aec"
```

This function corresponds with the standard semantics:

```
lemma eval_models:
  assumes "t ∈ Clock.jkbpC" and "clock_simAbs ec = Clock.sim_equiv_class a t"
  assumes "subjective a φ"
  assumes "s ∈ ODList.toSet (snd ec)"
  shows "eval envVal envObs ec φ ⟷ clock_repMC (ODList.toSet (fst ec)), s ⊨ φ"
```

Simulated actions

We can compute the actions enabled for a from a common equivalence class and a subjective equivalence class for agent a:

```
definition (in -)
  clock_simAction :: "('a, 'p, 'aAct) JKBP ⇒ (('s :: linorder) ⇒ 'p ⇒ bool)
                  ⇒ ('a ⇒ 's ⇒ 'obs) ⇒ 'a ⇒ 's clock_simWorldsRep ⇒ 'aAct list"
where "clock_simAction jkbp envVal envObs ≡ λa (Y, X).
  [ action gc. gc ← jkbp a, eval envVal envObs (Y, X) (guard gc) ]"
```

Using `eval_models`, we can relate clock_simAction to jAction. Firstly, clock_simAction behaves the same as jAction using the clock_repMC structure:

```
lemma clock_simAction_jAction:
  assumes "t ∈ Clock.jkbpC" and "clock_simAbs ec = Clock.sim_equiv_class a t"
  shows "set (clock_simAction a ec)
       = set (jAction (clock_repMC (ODList.toSet (fst ec))) (tLast t) a)"
```

We can connect the agent's choice of actions on the clock_repMC structure to those on the Clock.MC structure via clock_simMCt t using our earlier results about actions being preserved by generated models and simulations.

```
lemma clock_simAction:
  assumes "t ∈ Clock.jkbpC" and "clock_simAbs ec = Clock.sim_equiv_class a t"
  shows "set (clock_simAction a ec) = set (jAction Clock.MC t a)"
```

Simulated transitions

We use the clock_trans function to determine the image of the set of commonly-held-possible states under the transition function, and also for the agent's subjective equivalence class.

```
definition (in -) clock_trans :: "('a :: linorder) odlist ⇒ ('a, 'p, 'aAct) JKBP
                          ⇒ (('s :: linorder) ⇒ 'eAct list)
                          ⇒ ('eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's)
                          ⇒ ('s ⇒ 'p ⇒ bool) ⇒ ('a ⇒ 's ⇒ 'obs)
                          ⇒ 's odlist ⇒ 's odlist ⇒ 's odlist"
where "clock_trans agents jkbp envAction envTrans envVal envObs ≡ λcec X.
  ODList.fromList (concat [ [ envTrans eact aact s .
    eact ← envAction s,
    aact ← listToFuns (λa. clock_simAction jkbp envVal envObs a
                            (cec, clock_knowledge envObs cec a s))
                            (toList agents) ] . s ← toList X ])"
```

The function listToFuns exhibits the isomorphism between `('a × 'b list) list` and `('a ⇒ 'b) list` for finite types `'a`.

We can show that the transition function works in both cases.

```
lemma clock_trans_common:
  assumes "t ∈ Clock.jkbpC" and "clock_simAbs ec = Clock.sim_equiv_class a t"
  shows "ODList.toSet (clock_trans (fst ec) (fst ec))
      = { s |t' s. t' ↝ s ∈ Clock.jkbpC ∧ tLength t' = tLength t }"

lemma clock_trans_agent:
  assumes "t ∈ Clock.jkbpC" and "clock_simAbs ec = Clock.sim_equiv_class a t"
  shows "ODList.toSet (clock_trans (fst ec) (snd ec))
      = { s |t' s. t' ↝ s ∈ Clock.jkbpC ∧ clock_jview a t' = clock_jview a t }"
```

As the clock semantics disregards paths we simply compute the successors of the temporal slice and partition that. Similarly the successors of the agent's subjective equivalence class tell us what the set of possible observations are.

```
definition (in -) clock_mkSuccs :: "('s :: linorder ⇒ 'obs) ⇒ 'obs ⇒ 's odlist
                                    ⇒ 's clock_simWorldsRep"
where "clock_mkSuccs envObs obs Y' ≡ (Y', ODList.filter (λs. envObs s = obs) Y')"
```

Finally we can define our transition function on simulated states:

```
definition (in -)
  clock_simTrans :: "('a :: linorder) odlist ⇒ ('a, 'p, 'aAct) JKBP
              ⇒ (('s :: linorder) ⇒ 'eAct list)
              ⇒ ('eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's)
              ⇒ ('s ⇒ 'p ⇒ bool) ⇒ ('a ⇒ 's ⇒ 'obs)
              ⇒ 'a ⇒ 's clock_simWorldsRep ⇒ 's clock_simWorldsRep list"
where
  "clock_simTrans agents jkbp envAction envTrans envVal envObs ≡ λa (Y, X).
     let X' = clock_trans agents jkbp envAction envTrans envVal envObs Y X;
         Y' = clock_trans agents jkbp envAction envTrans envVal envObs Y Y
      in [ clock_mkSuccs (envObs a) obs Y' .
             obs ← map (envObs a) (toList X') ]"
```

Showing that this respects the property asked of it by the Algorithm locale is straightforward:

```
lemma clock_simTrans:
  assumes "t ∈ Clock.jkbpC" and "clock_simAbs ec = Clock.sim_equiv_class a t"
  shows "clock_simAbs ` set (clock_simTrans a ec)
     = { Clock.sim_equiv_class a (t' ↝ s)
          |t' s. t' ↝ s ∈ Clock.jkbpC ∧ clock_jview a t' = clock_jview a t }"
end
```

Maps

As mentioned above, the canonicity of our ordered, distinct list representation of automaton states allows us to use them as keys in a digital trie; a value of type `('key, 'val) trie` maps keys of type `'key list` to values of type `'val`.

In this specific case we track automaton transitions using a two-level structure mapping sets of states to an association list mapping observations to sets of states. For actions automaton states map directly to agent actions.

```
type_synonym ('s, 'obs) clock_trans_trie
        = "('s, ('s, ('obs, 's clock_simWorldsRep) mapping) trie) trie"
type_synonym ('s, 'aAct) clock_acts_trie = "('s, ('s, 'aAct) trie) trie"
```

We define two records of map operations acts_MapOps and trans_MapOps for these types and show that they satisfy the MapOps predicate (§3.6.6). Discharging the obligations in the Algorithm locale is routine, leaning on the work of Lammich and Lochbihler (2010).

Locale instantiation

Finally we assemble the algorithm and discharge the proof obligations.

```
sublocale FiniteLinorderEnvironment < Clock!: Algorithm jkbp envInit envAction
  envTrans envVal clock_jview envObs clock_jviewInit clock_jviewIncr clock_sim
  clock_simRels clock_simVal clock_simAbs clock_simObs clock_simInit clock_simTrans
  clock_simAction acts_MapOps trans_MapOps
```

Explicitly, the algorithm for this case is:

```
definition "mkClockAuto ≡ λagents jkbp envInit envAction envTrans envVal envObs.
  mkAlgAuto acts_MapOps
            trans_MapOps
            (clock_simObs envObs)
            (clock_simInit envInit envObs)
            (clock_simTrans agents jkbp envAction envTrans envVal envObs)
            (clock_simAction jkbp envVal envObs)
            (λa. map (clock_simInit envInit envObs a ∘ envObs a) envInit)"

lemma (in FiniteLinorderEnvironment) mkClockAuto_implements:
"Clock.implements (mkClockAuto agents jkbp envInit envAction envTrans envVal envObs)"
```

We discuss the clock semantics further in §6.2.1.

3.7.2 The synchronous perfect-recall view

The synchronous perfect-recall (SPR) view records all observations the agent has made on a given trace. It is the canonical full-information synchronous view, and simply maintains a list of all observations made on the trace:

```
definition (in Environment) spr_jview :: "('a, 's, 'obs Trace) JointView" where
  "spr_jview a = tMap (envObs a)"
```

The corresponding incremental view appends a new observation to the existing ones:

```
definition (in Environment) spr_jviewInit :: "'a ⇒ 'obs ⇒ 'obs Trace" where
  "spr_jviewInit ≡ λa obs. tInit obs"

definition (in Environment) spr_jviewIncr :: "'a ⇒ 'obs ⇒ 'obs Trace ⇒ 'obs Trace"
where "spr_jviewIncr ≡ λa obs' tobs. tobs ⇝ obs'"

sublocale Environment
  < SPR!: IncrEnvironment jkbp envInit envAction envTrans envVal
                                spr_jview envObs spr_jviewInit spr_jviewIncr
```

van der Meyden (1996b, Theorem 5) showed that finite-state implementations do not always exist with respect to the SPR view, and so we consider three special cases:

§3.7.3 where there is a single agent;

§3.7.4 when the protocols of the agents are deterministic and communicate by broadcast; and

§3.7.5 when the agents use non-deterministic protocols and broadcast.

These cases do overlap but none is wholly contained in another.

3.7.3 Perfect recall for a single agent

We capture our expectations of a single-agent scenario in the following locale:

```
locale FiniteSingleAgentEnvironment =
  FiniteEnvironment jkbp envInit envAction envTrans envVal envObs
    for jkbp :: "('a, 'p, 'aAct) JKBP"
    and envInit :: "('s :: {finite, linorder}) list"
    and envAction :: "'s ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's"
    and envVal :: "'s ⇒ 'p ⇒ bool"
    and envObs :: "'a ⇒ 's ⇒ 'obs"
+ fixes agent :: "'a" assumes envSingleAgent: "a = agent"
```

For algorithmic reasons we assume that the set of states is finite and linearly ordered. The sole agent is named `agent`.

Our simulation is similar to that for the clock semantics of §3.7.1: it records the set of worlds that the agent considers possible relative to a trace and the SPR view. The difference is that it is path sensitive.

```
context FiniteSingleAgentEnvironment
begin

definition spr_abs :: "'s Trace ⇒ 's set" where
 "spr_abs t ≡ tLast ` { t' ∈ SPR.jkbpC . spr_jview agent t' = spr_jview agent t }"
```

```
type_synonym (in -) 's spr_simWorlds = "'s set × 's"

definition spr_sim :: "'s Trace ⇒ 's spr_simWorlds" where
  "spr_sim ≡ λt. (spr_abs t, tLast t)"
```

The corresponding $S5_n$ structure relates worlds for `agent` if the sets of possible states coincide and the observation of the final states is the same. Propositions are evaluated at the final state.

```
definition spr_simRels :: "'a ⇒ ('s spr_simWorlds × 's spr_simWorlds) set" where
  "spr_simRels ≡ λa. { ((U, u), (V, v)) |U u V v.
                       U = V ∧ {u, v} ⊆ U ∧ envObs a u = envObs a v }"

definition spr_simVal :: "'s spr_simWorlds ⇒ 'p ⇒ bool" where
  "spr_simVal ≡ envVal ∘ snd"

abbreviation spr_simMC :: "('a, 'p, 's spr_simWorlds) KripkeStructure" where
  "spr_simMC ≡ mkKripke (spr_sim ` SPR.jkbpC) spr_simRels spr_simVal"

lemma spr_sim: "sim SPR.MC spr_simMC spr_sim"
end

sublocale FiniteSingleAgentEnvironment
  < SPRsingle!: SimIncrEnvironment jkbp envInit envAction envTrans envVal spr_jview
              envObs spr_jviewInit spr_jviewIncr spr_sim spr_simRels spr_simVal
```

Representations

As in §3.7.1, we quotient `'s spr_simWorlds` by `spr_simRels`. In this single-agent case, the element of this quotient corresponding to cononical trace `t` is isomorphic to the set of states that are possible given the sequence of observations made by `agent` on `t`. Therefore we have a simple representation:

```
context FiniteSingleAgentEnvironment
begin

type_synonym (in -) 's spr_simWorldsRep = "'s odlist"
```

It is very easy to map these representations back to simulated equivalence classes:

```
definition spr_simAbs :: "'s spr_simWorldsRep ⇒ 's spr_simWorlds set" where
  "spr_simAbs ≡ λss. { (toSet ss, s) |s. s ∈ toSet ss }"
```

This time our representation is unconditionally canonical:

```
lemma spr_simAbs_inj: "inj spr_simAbs"
```

We again make use of the following Kripke structure, where the worlds are the final states of the subset of the temporal slice that `agent` believes possible:

```
definition spr_repRels :: "'a ⇒ ('s × 's) set" where
  "spr_repRels ≡ λa. { (s, s'). envObs a s' = envObs a s }"

abbreviation spr_repMC :: "'s set ⇒ ('a, 'p, 's) KripkeStructure" where
  "spr_repMC ≡ λX. mkKripke X spr_repRels envVal"
```

Similarly we show that this Kripke structure is adequate by introducing an intermediate structure and connecting them all with a tower of simulations:

```
abbreviation spr_jkbpCSt :: "'s Trace ⇒ 's spr_simWorlds set" where
  "spr_jkbpCSt t ≡ SPRsingle.sim_equiv_class agent t"

abbreviation spr_simMCt :: "'s Trace ⇒ ('a, 'p, 's spr_simWorlds) KripkeStructure"
where "spr_simMCt t ≡ mkKripke (spr_jkbpCSt t) spr_simRels spr_simVal"

definition spr_repSim :: "'s spr_simWorlds ⇒ 's" where "spr_repSim ≡ snd"

lemma spr_repSim: "sim (spr_simMCt t) ((spr_repMC ∘ fst) (spr_sim t)) spr_repSim"
```

As before, the following sections discharge the requirements of the Algorithm locale of Figure 3.3.

Initial states

The initial states of the automaton for agent is simply the partition of envInit under agent's observation.

```
definition (in -) spr_simInit :: "('s :: linorder) list ⇒ ('a ⇒ 's ⇒ 'obs)
                                  ⇒ 'a ⇒ 'obs ⇒ 's spr_simWorldsRep"
where "spr_simInit envInit envObs ≡ λa iobs.
  ODList.fromList [ s. s ← envInit, envObs a s = iobs ]"

lemma spr_simInit:
  assumes "iobs ∈ envObs a ` set envInit"
  shows "spr_simAbs (spr_simInit a iobs)
       = spr_sim ` { t' ∈ SPR.jkbpC. spr_jview a t' = spr_jviewInit a iobs }"
```

Simulated observations

As the agent makes the same observation on the entire equivalence class, we arbitrarily choose the first element of the representation:

```
definition (in -) spr_simObs :: "('a ⇒ 's ⇒ 'obs)
                                 ⇒ 'a ⇒ ('s :: linorder) spr_simWorldsRep ⇒ 'obs"
where "spr_simObs envObs ≡ λa. envObs a ∘ ODList.hd"

lemma spr_simObs:
  assumes "t ∈ SPR.jkbpC" and "spr_simAbs ec = SPRsingle.sim_equiv_class a t"
  shows "spr_simObs a ec = envObs a (tLast t)"
```

Evaluation

As the single-agent case is much simpler than the multi-agent ones we define a specialised evaluation function. Intuitively eval yields the subset of X where the formula holds, where X is a representation of a canonical equivalence class for agent.

```
fun (in -) eval :: "(('s :: linorder) ⇒ 'p ⇒ bool)
                ⇒ 's odlist ⇒ ('a, 'p) Kform ⇒ 's odlist"
where
  "eval val X (Kprop p)    = ODList.filter (λs. val s p) X"
| "eval val X (Knot φ)      = ODList.difference X (eval val X φ)"
| "eval val X (Kand φ ψ)  = ODList.intersect (eval val X φ) (eval val X ψ)"
| "eval val X (K_a φ)       = (if eval val X φ = X then X else ODList.empty)"
| "eval val X (C_as φ)      = (if as = [] ∨ eval val X φ = X then X else ODList.empty)"
```

In general this is less efficient than the tableau approach of Fagin et al. (1995, Proposition 3.2.1), which labels all states with all formulas. However it is often the case that the set of relevant worlds is much smaller than the set of all system states.

Showing that this corresponds with the standard models relation is routine.

```
lemma eval_models:
  assumes ec: "spr_simAbs ec = SPRsingle.sim_equiv_class agent t"
  assumes subj: "subjective agent φ"
  assumes s: "s ∈ toSet ec"
  shows "toSet (eval envVal ec φ) ≠ {} ⟷ spr_repMC (toSet ec), s ⊨ φ"
```

Simulated actions

The enabled actions on a canonical equivalence class X are those with satisfied guards:

```
definition (in -) spr_simAction :: "('a, 'p, 'aAct) KBP
         ⇒ (('s :: linorder) ⇒ 'p ⇒ bool) ⇒ 'a ⇒ 's spr_simWorldsRep ⇒ 'aAct list"
where "spr_simAction kbp envVal ≡ λa X.
  [ action gc. gc ← kbp, eval envVal X (guard gc) ≠ ODList.empty ]"
```

The key lemma relates the agent's behaviour on an equivalence class to that on its representation:

```
lemma spr_simAction_jAction:
  assumes "t ∈ SPR.jkbpC" and "spr_simAbs ec = SPRsingle.sim_equiv_class agent t"
  shows "set (spr_simAction agent ec)
       = set (jAction (spr_repMC (toSet ec)) (tLast t) agent)"
```

We satisfy the Algorithm locale by chaining the above simulations.

```
lemma spr_simAction:
  assumes "t ∈ SPR.jkbpC" and "spr_simAbs ec = SPRsingle.sim_equiv_class a t"
  shows "set (spr_simAction a ec) = set (jAction SPR.MC t a)"
```

Simulated transitions

We can compute the possible successor states of a canonical equivalence class X:

```
definition (in -) spr_trans :: "('a, 'p, 'aAct) KBP ⇒ ('s ⇒ 'eAct list)
                          ⇒ ('eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's) ⇒ ('s ⇒ 'p ⇒ bool)
                          ⇒ 'a ⇒ ('s :: linorder) spr_simWorldsRep ⇒ 's list"
where "spr_trans kbp envAction envTrans val ≡ λa X.
  [ envTrans eact (λa'. aact) s .
    s ← toList X, eact ← envAction s, aact ← spr_simAction kbp val a X ]"
```

Using this function we can determine the set of possible successor equivalence classes from X:

```
abbreviation (in -) envObs_rel :: "('s ⇒ 'obs) ⇒ ('s × 's ⇒ bool)" where
  "envObs_rel ≡ λenvObs (s, s'). envObs s' = envObs s"

definition (in -) spr_simTrans :: "('a, 'p, 'aAct) KBP
  ⇒ (('s::linorder) ⇒ 'eAct list) ⇒ ('eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's)
  ⇒ ('s ⇒ 'p ⇒ bool) ⇒ ('a ⇒ 's ⇒ 'obs)
  ⇒ 'a ⇒ 's spr_simWorldsRep ⇒ 's spr_simWorldsRep list"
where "spr_simTrans kbp envAction envTrans val envObs ≡ λa X.
  map ODList.fromList (partition (envObs_rel (envObs a))
                                 (spr_trans kbp envAction envTrans val a X))"
```

The partition function splits a list into equivalence classes under the given equivalence relation.

The property asked for by the Algorithm locale is as follows.

```
lemma spr_simTrans:
  assumes "t ∈ SPR.jkbpC" and "spr_simAbs ec = SPRsingle.sim_equiv_class a t"
  shows "spr_simAbs ` set (spr_simTrans a ec)
      = { SPRsingle.sim_equiv_class a (t' ↝ s)
          |t' s. t' ↝ s ∈ SPR.jkbpC ∧ spr_jview a t' = spr_jview a t}"

end
```

Maps

As in §3.7.1, we use a pair of tries and an association list to handle the automata representation. Recall that the keys of these tries are lists of system states.

```
type_synonym ('s, 'obs) spr_trans_trie = "('s, ('obs, 's odlist) mapping) trie"
type_synonym ('s, 'aAct) spr_acts_trie = "('s, ('s, 'aAct) trie) trie"
```

Locale instantiation

The above is sufficient to instantiate the Algorithm locale.

```
sublocale FiniteSingleAgentEnvironment
  < SPRsingle!: Algorithm jkbp envInit envAction envTrans envVal spr_jview envObs
```

```
spr_jviewInit spr_jviewIncr spr_sim spr_simRels spr_simVal
spr_simAbs spr_simObs spr_simInit spr_simTrans spr_simAction
trie_odlist_MapOps trans_MapOps
```

We use this theory to construct a solution to the robot of §2 in §3.8.1.

3.7.4 Perfect recall in deterministic broadcast environments

It is well known that simultaneous broadcast has the effect of making information *common knowledge*; roughly put, all agents simultaneously learn the same thing from the broadcast, and so the relation amongst the agents' states of knowledge never becomes more complex than it was before (Fagin et al. 1995, Chapter 6). For this reason we might hope to find finite-state implementations of JKBPs in such environments. However van der Meyden (1996b, §7) showed that we need to further constrain the scenario. Here we require that for each canonical trace the JKBP prescribes at most one action. In practice this constraint is easier to verify than the circularity would suggest; we return to this point at the end of this section.

We encode our expectations in the FiniteBroadcastEnvironment locale of Figure 3.5. The broadcast is modelled by having all agents make the same common observation of the shared state of type `'es`. We also allow each agent to maintain a private state of type `'ps`; that other agents cannot directly influence or observe it is enforced by the constraint `envTrans` and the definition of `envObs`. In contrast we allow the environment's protocol envAction to be non-deterministic and a function of the entire system state, including private states.

```
context FiniteDetBroadcastEnvironment
begin
```

We seek a suitable simulation by considering what determines an agent's knowledge. Intuitively any trace that is relevant to the agents' states of knowledge with respect to `t ∈ jkbpC` needs to have the same common observation as `t`. Clearly this is an abstraction of the SPR jview

```
definition tObsC :: "('a, 'es, 'as) BEState Trace ⇒ 'cobs Trace" where
  "tObsC ≡ tMap (envObsC ∘ es)"

lemma spr_jview_tObsC:
  assumes "spr_jview a t = spr_jview a t'"
  shows "tObsC t = tObsC t'"
```

Unlike the single-agent case of §3.7.3, it is not sufficient for a simulation to record only the final states; it may be that the initial states may contain information that is not common knowledge. We therefore relate the final state of a trace to its initial state.

```
definition tObsC_abs :: "('a, 'es, 'as) BEState Trace
            ⇒ (('a, 'es, 'as) BEState × ('a, 'es, 'as) BEState) set"
where "tObsC_abs t ≡ { (tFirst t', tLast t')
                      |t'. t' ∈ SPR.jkbpC ∧ tObsC t' = tObsC t}"
```

```
record ('a, 'es, 'ps) BEState =
  es :: "'es"
  ps :: "('a × 'ps) odlist"

locale FiniteDetBroadcastEnvironment =
  Environment jkbp envInit envAction envTrans envVal envObs
    for jkbp :: "'a ⇒ ('a :: {finite, linorder}, 'p, 'aAct) KBP"
    and envInit
          :: "('a, 'es :: {finite, linorder}, 'as :: {finite, linorder}) BEState list"
    and envAction :: "('a, 'es, 'as) BEState ⇒ 'eAct list"
    and envTrans :: "'eAct ⇒ ('a ⇒ 'aAct)
                       ⇒ ('a, 'es, 'as) BEState ⇒ ('a, 'es, 'as) BEState"
    and envVal :: "('a, 'es, 'as) BEState ⇒ 'p ⇒ bool"
    and envObs :: "'a ⇒ ('a, 'es, 'as) BEState ⇒ ('cobs × 'as option)"

+ fixes agents :: "'a odlist"
  fixes envObsC :: "'es ⇒ 'cobs"
  defines "envObs a s ≡ (envObsC (es s), ODList.lookup (ps s) a)"
  assumes agents: "ODList.toSet agents = UNIV"
  assumes envTrans: "∀s s' a eact eact' aact aact'.
            ODList.lookup (ps s) a = ODList.lookup (ps s') a ∧ aact a = aact' a
             ⟶ ODList.lookup (ps (envTrans eact aact s)) a
               = ODList.lookup (ps (envTrans eact' aact' s')) a"
  assumes jkbpDet: "∀a. ∀t ∈ SPR.jkbpC. length (jAction SPR.MC t a) ≤ 1"
```

Figure 3.5: Finite broadcast environments with a deterministic JKBP.

We use the following record to represent the worlds of the simulated Kripke structure:

```
record ('a, 'es, 'as) spr_simWorld =
  sprFst :: "('a, 'es, 'as) BEState"
  sprLst :: "('a, 'es, 'as) BEState"
  sprRel :: "(('a, 'es, 'as) BEState × ('a, 'es, 'as) BEState) set"
```

The simulation of a trace t ∈ jkbpC records its initial and final states, and the relation between initial and final states of all commonly-plausible traces:

```
definition spr_sim :: "('a, 'es, 'as) BEState Trace ⇒ ('a, 'es, 'as) spr_simWorld"
where "spr_sim ≡ λt. ⦇ sprFst = tFirst t, sprLst = tLast t, sprRel = tObsC_abs t ⦈"
```

We relate two worlds in the associated Kripke structure if the agent's observations on the the first and last states correspond, and both have the same common observation relation.

```
definition spr_simRels :: "'a ⇒ (('a, 'es, 'as) spr_simWorld
                              × ('a, 'es, 'as) spr_simWorld) set"
where "spr_simRels ≡ λa. { (s, s') |s s'.
                             envObs a (sprFst s) = envObs a (sprFst s')
                           ∧ envObs a (sprLst s) = envObs a (sprLst s')
                           ∧ sprRel s = sprRel s' }"

definition spr_simVal :: "('a, 'es, 'as) spr_simWorld ⇒ 'p ⇒ bool" where
  "spr_simVal ≡ envVal ∘ sprLst"
```

```
abbreviation "spr_simMC ≡ mkKripke (spr_sim ‘ SPR.jkbpC) spr_simRels spr_simVal"
```

All simulation properties are easy to show for spr_sim except for reverse simulation. The latter follows from the fact that for two traces with the same common observations where agent a makes the same observation on their initial states, then a's private states on the two traces are identical.

```
lemma spr_jview_det_ps:
  assumes "{t, t'} ⊆ SPR.jkbpC"
  assumes "tObsC t = tObsC t'"
  assumes "envObs a (tFirst t) = envObs a (tFirst t')"
  shows "tMap (λs. ODList.lookup (ps s) a) t = tMap (λs. ODList.lookup (ps s) a) t'"
```

The proof proceeds by simultaneous induction over t and t', appealing to the jkbpDet locale assumption, the definition of envObs and the constraint envTrans.

```
lemma spr_sim: "sim SPR.MC spr_simMC spr_sim"
end

sublocale FiniteDetBroadcastEnvironment
  < SPRdet!: SimIncrEnvironment jkbp envInit envAction envTrans envVal spr_jview
             envObs spr_jviewInit spr_jviewIncr spr_sim spr_simRels spr_simVal
```

Representations

As before we canonically represent the quotient of the simulated worlds under spr_simRels using ordered, distinct lists. In particular, we use the type ('a × 'a) odlist (abbreviated 'a odrelation) to canonically represent relations.

```
context FiniteDetBroadcastEnvironment
begin

type_synonym (in -) ('a, 'es, 'as) spr_simWorldsECRep
  = "('a, 'es, 'as) BEState odrelation"
type_synonym (in -) ('a, 'es, 'as) spr_simWorldsRep
  = "('a, 'es, 'as) spr_simWorldsECRep × ('a, 'es, 'as) spr_simWorldsECRep"
```

We can abstract such a representation into a set of simulated equivalence classes:

```
definition spr_simAbs :: "('a, 'es, 'as) spr_simWorldsRep
                          ⇒ ('a, 'es, 'as) spr_simWorld set"
where "spr_simAbs ≡ λ(cec, aec). { ⦇ sprFst = s, sprLst = s', sprRel = toSet cec ⦈
                                    |s s'. (s, s') ∈ toSet aec }"
```

For a representation X of the simulated equivalence class for t ∈ jkbpC, we can decompose the abstraction spr_simAbs X in terms of tObsC_abs t and the following function agent_abs t:

```
definition agent_abs :: "'a ⇒ ('a, 'es, 'as) BEState Trace
```

```
                        ⇒ ((’a, ’es, ’as) BEState × (’a, ’es, ’as) BEState) set"
where "agent_abs a t ≡ { (tFirst t’, tLast t’)
                          |t’. t’ ∈ SPR.jkbpC ∧ spr_jview a t’ = spr_jview a t }"
```

This representation is canonical on the domain of interest (though not in general):

```
lemma spr_simAbs_inj_on: "inj_on spr_simAbs { x . spr_simAbs x ∈ SPRdet.jkbpSEC }"
```

Later we use a Kripke structure constructed over `tObsC_abs t` for some `t ∈ jkbpC`.

```
type_synonym (in -) (’a, ’es, ’as) spr_simWorlds
  = "(’a, ’es, ’as) BEState × (’a, ’es, ’as) BEState"

definition (in -)
  spr_repRels :: "(’a ⇒ (’a, ’es, ’as) BEState ⇒ ’cobs × ’as option)
                 ⇒ ’a ⇒ ((’a, ’es, ’as) spr_simWorlds
                    × (’a, ’es, ’as) spr_simWorlds) set"
where "spr_repRels envObs ≡ λa.
  { ((u, v), (u’, v’)) . envObs a u = envObs a u’ ∧ envObs a v = envObs a v’ }"

definition spr_repVal :: "(’a, ’es, ’as) spr_simWorlds ⇒ ’p ⇒ bool" where
  "spr_repVal ≡ envVal ∘ snd"

abbreviation spr_repMC :: "((’a, ’es, ’as) BEState × (’a, ’es, ’as) BEState) set
                         ⇒ (’a, ’p, (’a, ’es, ’as) spr_simWorlds) KripkeStructure"
where "spr_repMC ≡ λtcobsR. mkKripke tcobsR (spr_repRels envObs) spr_repVal"
```

As before we show that this Kripke structure is adequate for a particular canonical trace `t` by showing that it simulates SPR.MC. We introduce an intermediate structure:

```
abbreviation
  spr_jkbpCSt :: "(’a, ’es, ’as) BEState Trace ⇒ (’a, ’es, ’as) spr_simWorld set"
where "spr_jkbpCSt t ≡ spr_sim ‘ { t’ . t’ ∈ SPR.jkbpC ∧ tObsC t = tObsC t’ }"

abbreviation spr_simMCt :: "(’a, ’es, ’as) BEState Trace
                         ⇒ (’a, ’p, (’a, ’es, ’as) spr_simWorld) KripkeStructure"
where "spr_simMCt t ≡ mkKripke (spr_jkbpCSt t) spr_simRels spr_simVal"

definition
  spr_repSim :: "(’a, ’es, ’as) spr_simWorld ⇒ (’a, ’es, ’as) spr_simWorlds"
where "spr_repSim ≡ λs. (sprFst s, sprLst s)"

lemma spr_repSim: "sim (spr_simMCt t) (spr_repMC (sprRel (spr_sim t))) spr_repSim"
```

We now define a set of functions that satisfy the Algorithm locale given the assumptions of the FiniteDetBroadcastEnvironment locale.

Initial states

The initial states for agent a given an initial observation iobs consist of the set of states that yield a common observation consonant with iobs paired with the set of states where a observes iobs:

```
definition (in -) spr_simInit :: "('a, 'es, 'as) BEState list ⇒ ('es ⇒ 'cobs)
                              ⇒ ('a ⇒ ('a, 'es, 'as) BEState ⇒ 'cobs × 'obs)
        ⇒ 'a ⇒ ('cobs × 'obs)
        ⇒ ('a :: linorder, 'es :: linorder, 'as :: linorder) spr_simWorldsRep"
where "spr_simInit envInit envObsC envObs ≡ λa iobs.
  (ODList.fromList [ (s, s). s ← envInit, envObsC (es s) = fst iobs ],
   ODList.fromList [ (s, s). s ← envInit, envObs a s = iobs ])"
```

```
lemma spr_simInit:
  assumes "iobs ∈ envObs a ‘ set envInit"
  shows "spr_simAbs (spr_simInit a iobs)
      = spr_sim ‘ { t’ ∈ SPR.jkbpC. spr_jview a t’ = spr_jviewInit a iobs }"
```

Simulated observations

Again we can choose any element of the representation of the simulated equivalence class:

```
definition (in -) spr_simObs :: "('es ⇒ 'cobs) ⇒ 'a::linorder
  ⇒ ('a, 'es::linorder, 'as::linorder) spr_simWorldsRep ⇒ 'cobs × 'as option"
where "spr_simObs envObsC ≡ λa. (λs. (envObsC (es s), ODList.lookup (ps s) a))
                              ∘ snd ∘ ODList.hd ∘ snd"
```

```
lemma spr_simObs:
  assumes "t ∈ SPR.jkbpC"
  assumes "spr_simAbs ec = SPRdet.sim_equiv_class a t"
  shows "spr_simObs a ec = envObs a (tLast t)"
```

Evaluation

As for the clock semantics (§3.7.1), we use the general evalation function evalS. Recall that propositions are used to filter the set of possible worlds X:

```
abbreviation (in -) spr_evalProp ::
  "(('a::linorder, 'es::linorder, 'as::linorder) BEState ⇒ 'p ⇒ bool)
  ⇒ ('a, 'es, 'as) BEState odrelation ⇒ 'p ⇒ ('a, 'es, 'as) BEState odrelation"
where "spr_evalProp envVal ≡ λX p. ODList.filter (λs. envVal (snd s) p) X"
```

The knowledge operation computes the subset of possible worlds cec that yield the same observation as s for agent a:

```
definition (in -) spr_knowledge ::
  "('a ⇒ ('a::linorder, 'es::linorder, 'as::linorder) BEState
        ⇒ 'cobs × 'as option) ⇒ ('a, 'es, 'as) BEState odrelation
```

```
  ⇒ 'a ⇒ ('a, 'es, 'as) spr_simWorlds ⇒ ('a, 'es, 'as) spr_simWorldsECRep"
where "spr_knowledge envObs cec ≡ λa s.
  ODList.fromList [ s' . s' ← toList cec, (s, s') ∈ spr_repRels envObs a ]"
```

Similarly the common knowledge operation computes the transitive closure (Sternagel and Thiemann 2011) of the union of the knowledge relations for the agents as:

```
definition (in -) spr_commonKnowledge ::
  "('a ⇒ ('a::linorder, 'es::linorder, 'as::linorder) BEState
          ⇒ 'cobs × 'as option) ⇒ ('a, 'es, 'as) BEState odrelation
  ⇒ 'a list ⇒ ('a, 'es, 'as) spr_simWorlds ⇒ ('a, 'es, 'as) spr_simWorldsECRep"
where "spr_commonKnowledge envObs cec ≡ λas s.
  let r = λa. ODList.fromList [ (s', s'') . s' ← toList cec, s'' ← toList cec,
                                              (s', s'') ∈ spr_repRels envObs a ];
      R = toList (ODList.big_union r as)
   in ODList.fromList (memo_list_trancl R s)"
```

We evaluate subjective knowledge formulas on representations of an equivalence class:

```
definition (in -) "eval envVal envObs ≡ λ(cec, X).
  evalS (spr_evalProp envVal) (spr_knowledge envObs cec)
        (spr_commonKnowledge envObs cec) X"
```

This function corresponds with the standard semantics:

```
lemma eval_models:
  assumes "t ∈ SPR.jkbpC" and "spr_simAbs ec = SPRdet.sim_equiv_class a t"
  assumes "subjective a φ"
  assumes "s ∈ toSet (snd ec)"
  shows "eval envVal envObs ec φ ⟷ spr_repMC (toSet (fst ec)), s ⊨ φ"
```

Simulated actions

From a common equivalence class and a subjective equivalence class for agent a, we can compute the actions enabled for a:

```
definition (in -) spr_simAction ::
      "('a, 'p, 'aAct) JKBP ⇒ (('a, 'es, 'as) BEState ⇒ 'p ⇒ bool)
    ⇒ ('a ⇒ ('a, 'es, 'as) BEState ⇒ 'cobs × 'as option) ⇒ 'a
    ⇒ ('a::linorder, 'es::linorder, 'as::linorder) spr_simWorldsRep ⇒ 'aAct list"
where "spr_simAction jkbp envVal envObs ≡ λa ec.
  [ action gc. gc ← jkbp a, eval envVal envObs ec (guard gc) ]"
```

Using the result about evaluation we can relate spr_simAction to jAction via spr_repMC:

```
lemma spr_action_jaction:
  assumes "t ∈ SPR.jkbpC" and "spr_simAbs ec = SPRdet.sim_equiv_class a t"
  shows "set (spr_simAction a ec)
      = set (jAction (spr_repMC (toSet (fst ec))) (tFirst t, tLast t) a)"
```

We connect the agent's choice of actions on the spr_repMC structure to those on SPR.MC using our earlier results about actions being preserved by generated models and simulations.

```
lemma spr_simAction:
  assumes "t ∈ SPR.jkbpC" and "spr_simAbs ec = SPRdet.sim_equiv_class a t"
  shows "set (spr_simAction a ec) = set (jAction SPR.MC t a)"
```

Simulated transitions

Simulating transitions is somewhat intricate. We begin by computing the successor relation of a given equivalence class X with respect to the common equivalence class cec:

```
abbreviation (in -) "spr_jAction jkbp envVal envObs cec s ≡ λa.
  spr_simAction jkbp envVal envObs a (cec, spr_knowledge envObs cec a s)"

definition (in -) spr_trans :: "'a odlist ⇒ ('a, 'p, 'aAct) JKBP
  ⇒ (('a::linorder, 'es::linorder, 'as::linorder) BEState ⇒ 'eAct list)
  ⇒ ('eAct ⇒ ('a ⇒ 'aAct) ⇒ ('a, 'es, 'as) BEState ⇒ ('a, 'es, 'as) BEState)
  ⇒ (('a, 'es, 'as) BEState ⇒ 'p ⇒ bool)
  ⇒ ('a ⇒ ('a, 'es, 'as) BEState ⇒ 'cobs × 'as option)
  ⇒ ('a, 'es, 'as) spr_simWorldsECRep ⇒ ('a, 'es, 'as) spr_simWorldsECRep
  ⇒ (('a, 'es, 'as) BEState × ('a, 'es, 'as) BEState) list"
where "spr_trans agents jkbp envAction envTrans envVal envObs ≡ λcec X.
  [ (initialS, succS) . (initialS, finalS) ← toList X, eact ← envAction finalS,
       succS ← [ envTrans eact aact finalS .
                     aact ← listToFuns (spr_jAction jkbp envVal envObs cec
                                                    (initialS, finalS))
                                          (toList agents) ] ]"
```

We split the result of this function according to the common observation and also agent a's observation, where a is the agent we are constructing the automaton for.

```
definition (in -) spr_simObsC :: "('es::linorder ⇒ 'cobs)
  ⇒ (('a::linorder, 'es, 'as::linorder) BEState × ('a, 'es, 'as) BEState) odlist
  ⇒ 'cobs"
where "spr_simObsC envObsC ≡ envObsC ∘ es ∘ snd ∘ ODList.hd"

abbreviation (in -) envObs_rel :: "(('a, 'es, 'as) BEState ⇒ 'cobs × 'as option)
  ⇒ (('a, 'es, 'as) spr_simWorlds × ('a, 'es, 'as) spr_simWorlds ⇒ bool)"
where "envObs_rel envObs ≡ λ(s, s'). envObs (snd s') = envObs (snd s)"
```

The above are combined in a function that yields the successor equivalence classes:

```
definition (in -) spr_simTrans :: "('a::linorder) odlist ⇒ ('a, 'p, 'aAct) JKBP
  ⇒ (('a, 'es::linorder, 'as::linorder) BEState ⇒ 'eAct list)
  ⇒ ('eAct ⇒ ('a ⇒ 'aAct) ⇒ ('a, 'es, 'as) BEState ⇒ ('a, 'es, 'as) BEState)
  ⇒ (('a, 'es, 'as) BEState ⇒ 'p ⇒ bool) ⇒ ('es ⇒ 'cobs)
  ⇒ ('a ⇒ ('a, 'es, 'as) BEState ⇒ 'cobs × 'as option) ⇒ 'a
```

```
  ⇒ ('a, 'es, 'as) spr_simWorldsRep ⇒ ('a, 'es, 'as) spr_simWorldsRep list"
where "spr_simTrans agents jkbp envAction envTrans envVal envObsC envObs ≡ λa ec.
  let aSuccs = spr_trans agents jkbp envAction envTrans envVal envObs
                          (fst ec) (snd ec);
      cec' = ODList.fromList (spr_trans agents jkbp envAction envTrans envVal envObs
                                                      (fst ec) (fst ec))
   in [ (ODList.filter (λs. envObsC (es (snd s)) = spr_simObsC envObsC aec') cec',
        aec')
      . aec' ← map ODList.fromList (partition (envObs_rel (envObs a)) aSuccs) ]"
```

Showing that spr_simTrans works requires a series of auxiliary lemmas that show we do in fact compute the correct successor equivalence classes. We elide the unedifying details, skipping straight to the lemma that the Algorithm locale expects:

```
lemma spr_simTrans:
  assumes "t ∈ SPR.jkbpC" and "spr_simAbs ec = SPRdet.sim_equiv_class a t"
  shows "spr_simAbs ` set (spr_simTrans a ec)
      = { SPRdet.sim_equiv_class a (t' ⇝ s)
          |t' s. t' ⇝ s ∈ SPR.jkbpC ∧ spr_jview a t' = spr_jview a t}"
end
```

The explicit-state approach sketched above is quite inefficient, and also some distance from the symbolic techniques we use in §6.2. However it does suffice to demonstrate the theory on the muddy children example in §3.8.2.

Maps

As always we use a pair of tries. The domain of these maps is the pair of relations.

```
type_synonym ('a, 'es, 'obs, 'as) trans_trie
  = "(('a, 'es, 'as) BEState, (('a, 'es, 'as) BEState,
      (('a, 'es, 'as) BEState, (('a, 'es, 'as) BEState,
       ('obs, ('a, 'es, 'as) spr_simWorldsRep) mapping) trie) trie) trie) trie"

type_synonym ('a, 'es, 'aAct, 'as) acts_trie
  = "(('a, 'es, 'as) BEState, (('a, 'es, 'as) BEState,
      (('a, 'es, 'as) BEState, (('a, 'es, 'as) BEState, 'aAct) trie) trie) trie) trie"
```

This suffices to placate the Algorithm locale.

```
sublocale FiniteDetBroadcastEnvironment
  < SPRdet!: Algorithm jkbp envInit envAction envTrans envVal spr_jview envObs
      spr_jviewInit spr_jviewIncr spr_sim spr_simRels spr_simVal spr_simAbs
      spr_simObs spr_simInit spr_simTrans spr_simAction acts_MapOps trans_MapOps
```

As we remarked earlier in this section, in general it may be difficult to establish the determinacy of a KBP as this property depends on the environment. However in many cases determinism is syntactically manifest in the JKBP as the guards are logically disjoint, independently of the knowledge subformulas. The following lemma generates the required proof obligations:

```
lemma (in PreEnvironmentJView) jkbpDetI:
  assumes "t ∈ jkbpC"
  assumes "∀a. distinct (map guard (jkbp a))"
  assumes "∀a gc gc'. gc ∈ set (jkbp a) ∧ gc' ∈ set (jkbp a) ∧ t ∈ jkbpC
    ⟶ guard gc = guard gc' ∨ ¬(MC, t ⊨ guard gc ∧ MC, t ⊨ guard gc')"
  shows "length (jAction MC t a) ≤ 1"
```

The scenario presented here is a variant of the broadcast environments treated by van der Meyden (1996b), which we cover in the next section.

3.7.5 Perfect recall in non-deterministic broadcast environments

For completeness we reproduce the results of van der Meyden (1996b) about non-deterministic KBPs in broadcast environments. The situation is described by the locale in Figure 3.6. Actions are now split into public and private components, where the private part influences the agents' private states, and the public part is broadcast and recorded in the system state. Moreover the protocol of the environment is a function of the environment state only. Once again an agent's view consists of the common observation and their private state. As the representations developed in the previous section are adequate for this case, we work more astractly here.

Our goal here is to instantiate the SimIncrEnvironment locale with respect to the assumptions made in the FiniteBroadcastEnvironment locale. This is similar to the previous section.

```
context FiniteBroadcastEnvironment
begin
```

As for the deterministic variant, we abstract traces using the common observation. Note that this now includes the public part of the agents' actions.

```
definition tObsC :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState Trace
                   ⇒ ('cobs × 'ePubAct × ('a ⇒ 'pPubAct)) Trace"
where "tObsC ≡ tMap (λs. (envObsC (es s), pubActs s))"
```

Similarly we introduce common and agent-specific abstraction functions:

```
definition tObsC_abs :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState Trace
                        ⇒ (('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
                         × ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState) set"
where "tObsC_abs t ≡ { (tFirst t', tLast t') |t'.
                        t' ∈ SPR.jkbpC ∧ tObsC t' = tObsC t }"

definition agent_abs :: "'a ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState Trace
               ⇒ (('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
                × ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState) set"
```

```
record ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState =
  es :: "'es"
  ps :: "'a ⇒ 'ps"
  pubActs :: "'ePubAct × ('a ⇒ 'pPubAct)"

locale FiniteBroadcastEnvironment =
  Environment jkbp envInit envAction envTrans envVal envObs
    for jkbp :: "('a :: finite, 'p, ('pPubAct :: finite × 'ps :: finite)) JKBP"
    and envInit
         :: "('a, 'ePubAct :: finite, 'es :: finite, 'pPubAct, 'ps) BEState list"
    and envAction :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
                  ⇒ ('ePubAct × 'ePrivAct) list"
    and envTrans :: "('ePubAct × 'ePrivAct)
                 ⇒ ('a ⇒ ('pPubAct × 'ps))
                 ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
                 ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState"
    and envVal :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState ⇒ 'p ⇒ bool"
    and envObs :: "'a ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
               ⇒ ('cobs × 'ps × ('ePubAct × ('a ⇒ 'pPubAct)))"

+ fixes envObsC :: "'es ⇒ 'cobs"
    and envActionES :: "'es ⇒ ('ePubAct × ('a ⇒ 'pPubAct))
                          ⇒ ('ePubAct × 'ePrivAct) list"
    and envTransES :: "('ePubAct × 'ePrivAct) ⇒ ('a ⇒ 'pPubAct)
                   ⇒ 'es ⇒ 'es"
  defines envObs_def: "envObs a ≡ (λs. (envObsC (es s), ps s a, pubActs s))"
      and envAction_def: "envAction s ≡ envActionES (es s) (pubActs s)"
      and envTrans_def:
           "envTrans eact aact s ≡ ⦇ es = envTransES eact (fst ∘ aact) (es s)
                                  , ps = snd ∘ aact
                                  , pubActs = (fst eact, fst ∘ aact) ⦈"
```

Figure 3.6: Finite broadcast environments with non-deterministic KBPs.

```
where "agent_abs a t ≡ { (tFirst t', tLast t') |t'.
                          t' ∈ SPR.jkbpC ∧ spr_jview a t' = spr_jview a t }"
```

The simulation is identical to that in the previous section:

```
record ('a, 'ePubAct, 'es, 'pPubAct, 'ps) SPRstate =
  sprFst :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState"
  sprLst :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState"
  sprRel :: "(('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
            × ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState) set"
definition spr_sim :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState Trace
                    ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) SPRstate"
where "spr_sim ≡ λt. ⦇ sprFst = tFirst t, sprLst = tLast t, sprRel = tObsC_abs t ⦈"
```

The Kripke structure over simulated traces is also the same:

```
definition spr_simRels :: "'a ⇒ (('a, 'ePubAct, 'es, 'pPubAct, 'ps) SPRstate
                             × ('a, 'ePubAct, 'es, 'pPubAct, 'ps) SPRstate) set"
where "spr_simRels ≡ λa. { (s, s') |s s'.
```

```
         envObs a (sprFst s) = envObs a (sprFst s')
       ∧ envObs a (sprLst s) = envObs a (sprLst s')
       ∧ sprRel s = sprRel s' }"
```

```
definition spr_simVal :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) SPRstate ⇒ 'p ⇒ bool"
where "spr_simVal ≡ envVal ∘ sprLst"
```

```
abbreviation "spr_simMC ≡ mkKripke (spr_sim ` SPR.jkbpC) spr_simRels spr_simVal"
```

As usual, showing that spr_sim is in fact a simulation is routine for all properties except for reverse simulation. For the latter we adapt the techniques of Lomuscio, van der Meyden, and Ryan (2000) (see also §A.5.2): we can show that, given t ∈ jkbpC, we can construct a trace t' ∈ jkbpC indistinguishable from t by agent a, based on the public actions, the common observation and a's private and initial states. We do this with a splicing operation:

```
definition sSplice :: "'a ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
                      ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
                      ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState"
where "sSplice a s s' ≡ s⦇ ps := (ps s)(a := ps s' a) ⦈"
```

The effect of sSplice a s s' is to update s with a's private state in s'. The key properties are that provided the common observation on s and s' are the same, then agent a's observation on sSplice a s s' is the same as at s', while everyone else's is the same as at s.

We hoist this operation pointwise to traces:

```
abbreviation tSplice :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState Trace ⇒ 'a
⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState Trace
⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState Trace" ("_ ⋈_ _" [50, 1000, 51] 50)
where "t ⋈a t' ≡ tZip (sSplice a) t t'"
```

The key properties are that after splicing, if t and t' have the same common observation, then so does $t \bowtie_a t'$, and for all agents $a' \neq a$, the view a' has of $t \bowtie_a t'$ is the same as it has of t, while for a it is the same as t'.

We can conclude that provided the two traces are initially indistinguishable to a, and not commonly distinguishable, then $t \bowtie_a t'$ is a canonical trace:

```
lemma tSplice_jkbpC:
  assumes tt': "{t, t'} ⊆ SPR.jkbpC"
  assumes init: "envObs a (tFirst t) = envObs a (tFirst t')"
  assumes tObsC: "tObsC t = tObsC t'"
  shows "t ⋈a t' ∈ SPR.jkbpC"
```

The proof is by simultaneous induction over t and t' and depends crucially on the public actions being recorded in the state and commonly observed.

```
lemma spr_sim: "sim SPR.MC spr_simMC spr_sim"
end
```

```
locale FiniteBroadcastEnvironmentIndependentInit =
  FiniteBroadcastEnvironment jkbp envInit envAction envTrans envVal envObs
                             envObsC envActionES envTransES
    for jkbp :: "('a::finite, 'p, ('pPubAct::{default,finite} × 'ps::finite)) JKBP"
    and envInit :: "('a, 'ePubAct :: {default, finite}, 'es :: finite,
                     'pPubAct, 'ps) BEState list"
    and envAction :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
                   ⇒ ('ePubAct × 'ePrivAct) list"
    and envTrans :: "('ePubAct × 'ePrivAct) ⇒ ('a ⇒ ('pPubAct × 'ps))
                   ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
                   ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState"
    and envVal :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState ⇒ 'p ⇒ bool"
    and envObs :: "'a ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState
                 ⇒ ('cobs × 'ps × ('ePubAct × ('a ⇒ 'pPubAct)))"
    and envObsC :: "'es ⇒ 'cobs"
    and envActionES :: "'es ⇒ ('ePubAct × ('a ⇒ 'pPubAct))
                           ⇒ ('ePubAct × 'ePrivAct) list"
    and envTransES :: "('ePubAct × 'ePrivAct) ⇒ ('a ⇒ 'pPubAct) ⇒ 'es ⇒ 'es"

+ fixes agents :: "'a list"
  fixes envInitES :: "'es list"
  fixes envInitPS :: "'a ⇒ 'ps list"
  defines envInit_def:
    "envInit ≡ [ ⦇ es = esf, ps = psf, pubActs = (default, λ_. default) ⦈
               . psf ← listToFuns envInitPS agents, esf ← envInitES ]"
  assumes agents: "set agents = UNIV" "distinct agents"
```

Figure 3.7: Finite broadcast environments with non-deterministic KBPs, where the initial private and environment states are independent.

```
sublocale FiniteBroadcastEnvironment
  < SPR!: SimIncrEnvironment jkbp envInit envAction envTrans envVal spr_jview
          envObs spr_jviewInit spr_jviewIncr spr_sim spr_simRels spr_simVal
```

Perfect recall in independently-initialised non-deterministic broadcast environments

If the private and environment parts of the initial states are independent we can reduce the state space of the construction of the previous section by working only with sets of states rather than relations. We capture this independence in the FiniteBroadcastEnvironmentIndependentInit locale shown in Figure 3.7 by asking that the initial states be the Cartesian product of possible private and environment states. As there are initially no public actions from the previous round, we use the default class to indicate that there is a fixed but arbitrary choice to be made here.

```
context FiniteBroadcastEnvironmentIndependentInit
begin

definition tObsC_ii_abs :: "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState Trace
                           ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState set"
where "tObsC_ii_abs t ≡ { tLast t' |t'. t' ∈ SPR.jkbpC ∧ tObsC t' = tObsC t }"
```

```
definition agent_ii_abs :: "'a ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState Trace
                         ⇒ ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState set"
where "agent_ii_abs a t ≡
  { tLast t' |t'. t' ∈ SPR.jkbpC ∧ spr_jview a t' = spr_jview a t }"
```

The simulation is similar to the single-agent case (§3.7.3); for a given canonical trace `t` it pairs the set of worlds that any agent considers possible with the final state of `t`:

```
type_synonym (in -) ('a, 'ePubAct, 'es, 'pPubAct, 'ps) SPRstate =
  "('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState set
 × ('a, 'ePubAct, 'es, 'pPubAct, 'ps) BEState"

definition "spr_ii_sim ≡ λt. (tObsC_ii_abs t, tLast t)"
definition "spr_ii_simRels ≡ λa.
  { (s, s') |s s'. envObs a (snd s) = envObs a (snd s') ∧ fst s = fst s' }"
definition "spr_ii_simVal ≡ envVal ∘ snd"
abbreviation
  "spr_ii_simMC ≡ mkKripke (spr_ii_sim ` SPR.jkbpC) spr_ii_simRels spr_ii_simVal"
```

That the simulation is adequate is shown in a similar way as earlier variants.

```
lemma spr_ii_sim: "sim SPR.MC spr_ii_simMC spr_ii_sim"
end

sublocale FiniteBroadcastEnvironmentIndependentInit
  < SPRii!: SimIncrEnvironment jkbp envInit envAction envTrans envVal spr_jview
        envObs spr_jviewInit spr_jviewIncr spr_ii_sim spr_ii_simRels spr_ii_simVal
```

3.8 Examples

We demonstrate the theory by using Isabelle's code generator to run it on two standard examples: the Robot from §2, and the classic Muddy Children puzzle.

3.8.1 The autonomous robot

Recall the autonomous robot of §2: we are looking for an implementation of the KBP:

$$\begin{array}{l}\textbf{do}\\ \quad [\!]\ \mathbf{K}_{\text{robot}}\ \mathsf{goal} \quad \to \mathsf{Halt}\\ \quad [\!]\ \neg\mathbf{K}_{\text{robot}}\ \mathsf{goal} \to \mathsf{Nothing}\\ \textbf{od}\end{array}$$

in an environment where positions are identified with the natural numbers, the robot's sensor is within one of the position, and the proposition goal is true when the position is in $\{2,3,4\}$. The robot is initially at position zero, and the effect of its Halt action is to cause it to instantaneously stop at its current position. A later Nothing action may allow the environment to move the robot further to the right. The Isabelle/HOL code for this scenario is shown in Figure 3.8.

To obtain a finite environment we truncate the number line at 5, which is intuitively sound for determinining the robot's behaviour due to the synchronous view, and the fact that if it reaches this rightmost position then it can never satisfy its objective. Running the Haskell code generated by Isabelle yields the automata shown in Figure 3.9 and Figure 3.10 for the clock and synchronous perfect recall semantics respectively after minimisation using Hopcroft's algorithm (Gries 1973).

The (inessential) labels on the states are an upper bound on the set of positions that the robot considers possible when it is in that state. Transitions are annotated with the observations yielded by the sensor. Double-circled states are those in which the robot performs the Halt action, the others Nothing. We observe that the synchronous perfect-recall view yields a "ratchet" protocol, i.e. if the robot learns that it is in the goal region then it halts for all time, and that it never overshoots the goal region. Conversely the clock semantics allows the robot to infinitely alternate its actions depending on the sensor reading. This is effectively the behaviour of the intuitive implementation that halts iff the sensor reads three or more.

We can also see that minimisation does not yield the smallest automata we could hope for; in particular there are several redundant states where the robot's prescribed behaviour is the same but its state of knowledge is different. This is because our implementations do not specify what happens on invalid observations, which we have modelled as errors instead of don't-cares, and these extraneous distinctions are preserved by minimisation. We discuss this further in §6.2.4.

3.8.2 The Muddy Children

Our first example of a multi-agent broadcast scenario is the classic Muddy Children puzzle, one of a class of puzzles that exemplify non-obvious reasoning about mutual states of knowledge. It goes as follows (Fagin et al. 1995, §1.1, Example 7.2.5):

> N children are playing together, k of whom get mud on their foreheads. Each can see the mud (or lack of mud) on the others' foreheads but not their own.
>
> A parental figure appears and says "At least one of you has mud on your forehead.", expressing something already known to each of them if $k > 1$.
>
> The parental figure then asks "Does any of you know whether you have mud on your own forehead?" over and over.
>
> Assuming the children are perceptive, intelligent, truthful and they answer simultaneously, what will happen?

This puzzle relies essentially on *synchronous public broadcasts* making particular facts *common knowledge*, and that agents are capable of the requisite logical inference.

```
type_synonym digit = "5 sat" — Saturated arithmetic

datatype Agent = robot
datatype EnvAct = Stay | MoveRight
datatype ObsErr = Left | On | Right
datatype Prop = halted | goal
datatype RobotAct = Nothing | Halt

type_synonym Halted = bool
type_synonym Obs = digit
type_synonym Pos = digit
type_synonym State = "Pos × Obs × Halted"

definition envInit :: "State list" where "envInit ≡ [(0, 0, False), (0, 1, False)]"

definition envAction :: "State ⇒ (EnvAct × ObsErr) list" where
  "envAction ≡ λ_. [ (x, y) . x ← [Stay, MoveRight], y ← [Left, On, Right] ]"

definition newObs :: "digit ⇒ ObsErr ⇒ digit" where
  "newObs pos obserr ≡
            case obserr of Left ⇒ pos - 1 | On ⇒ pos | Right ⇒ pos + 1"

definition
  envTrans :: "EnvAct × ObsErr ⇒ (Agent ⇒ RobotAct) ⇒ State ⇒ State"
where
  "envTrans ≡ λ(move, obserr) aact (pos, obs, halted).
    if halted then (pos, newObs pos obserr, halted)
          else case aact robot of
                Nothing ⇒ (case move of
                          Stay ⇒ (pos, newObs pos obserr, False)
                        | MoveRight ⇒ (pos + 1, newObs (pos + 1) obserr, False))
              | Halt ⇒ (pos, newObs pos obserr, True)"

definition envObs :: "State ⇒ Obs" where
  "envObs ≡ λ(pos, obs, halted). obs"

definition envVal :: "State ⇒ Prop ⇒ bool" where
  "envVal ≡ λ(pos, obs, halted) p.
     case p of halted ⇒ halted
            | goal ⇒ 2 ≤ pos ∧ pos ≤ (4 :: 5 sat)"

definition kbp :: "(Agent, Prop, RobotAct) KBP" where
  "kbp ≡ [ ⦇ guard = K_robot (Kprop goal),        action = Halt ⦈,
           ⦇ guard = Knot (K_robot (Kprop goal)), action = Nothing ⦈ ]"

interpretation Robot!:
  Environment "λ_. kbp" envInit envAction envTrans envVal "λ_. envObs"
interpretation Robot_Clock!: FiniteLinorderEnvironment
 "λ_. kbp" envInit envAction envTrans envVal "λ_. envObs" "ODList.fromList [robot]"
definition
  robot_ClockAlg :: "Agent ⇒ (digit, RobotAct, State odlist × State odlist) Protocol"
where "robot_ClockAlg ≡ mkClockAuto (ODList.fromList [robot]) (λ_. kbp) envInit
                                    envAction envTrans envVal (λ_. envObs)"

theorem (in FiniteLinorderEnvironment) "Robot.Clock.implements robot_ClockAlg"
```

Figure 3.8: The Isabelle/HOL definitions for the Robot example using the Clock view.

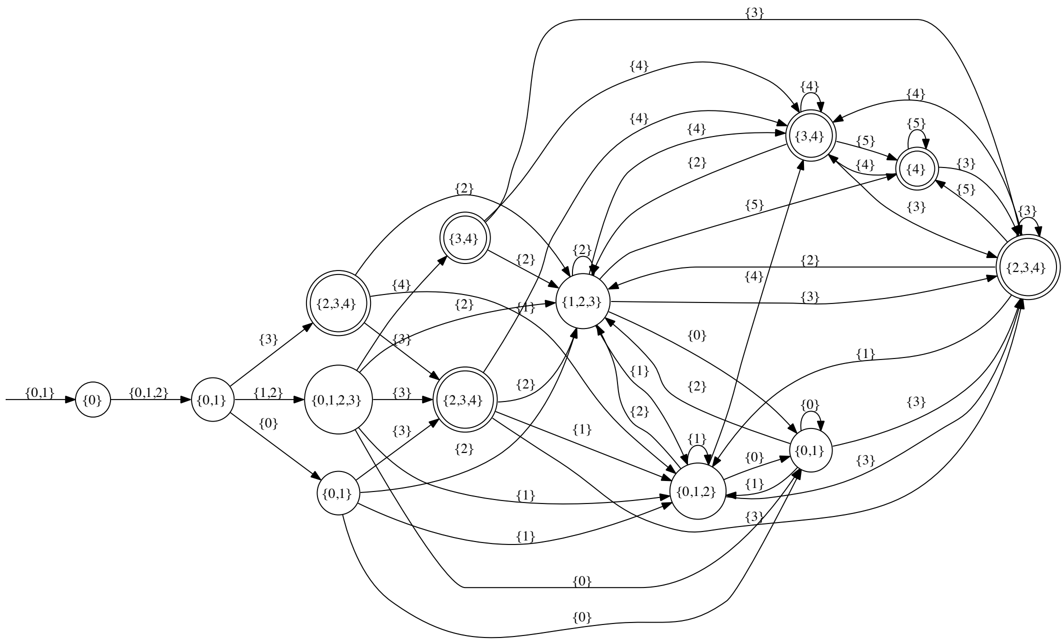


Figure 3.9: The implementation of the robot using the clock semantics.

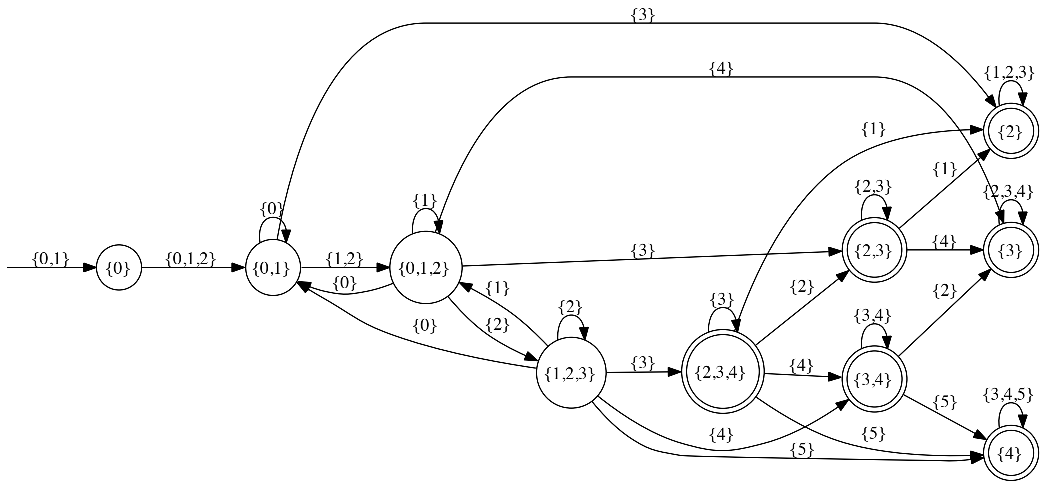


Figure 3.10: The implementation of the robot using the SPR semantics.

As the parental figure has complete knowledge of the situation, we integrate her behaviour into the environment. Each agent child_i reasons with the following KBP:

do
$\square\ \hat{\mathbf{K}}_{\mathrm{child}_i} \mathrm{muddy}_i \quad \rightarrow$ Say "I know if my forehead is muddy"
$\square\ \neg\hat{\mathbf{K}}_{\mathrm{child}_i} \mathrm{muddy}_i \rightarrow$ Say nothing
od

where $\hat{\mathbf{K}}_a\varphi$ abbreviates $\mathbf{K}_a\varphi \vee \mathbf{K}_a\neg\varphi$.

We use the SPR algorithm of §3.7.4 as this protocol is deterministic.

The model records a child's initial observation of the mother's pronouncement and the muddiness of the other children in her initial private state, and these states are not changed by `envTrans`. The recurring common observation is all of the children's public responses to the mother's questions. Being able to distinguish these observations is crucial to making this a broadcast scenario.

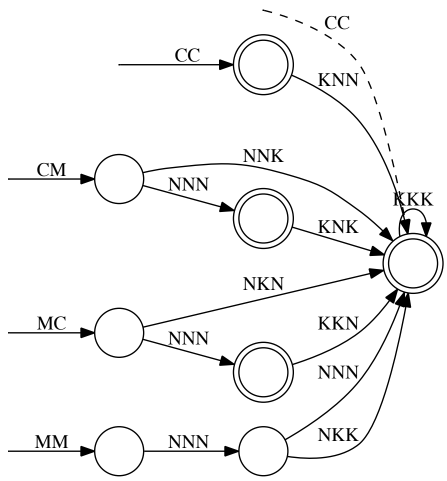


Figure 3.11: The protocol of child_0.

Running the algorithm for three children and minimising using Hopcroft's algorithm yields the automaton in Figure 3.11 for child_0. The initial transitions are labelled with the initial observation, i.e., the cleanliness "C" or muddiness "M" of the other two children. The dashed initial transition covers the case where everyone is clean; in the others the mother has announced that someone is dirty. Later transitions simply record the actions performed by each of the agents, where "K" is the first action in the above KBP, and "N" the second. Double-circled states are those in which child_0 knows whether she is muddy, and single-circled where she does not.

In essence the child counts the number of muddy foreheads she sees and waits that many rounds before announcing that she knows.

Note that a solution to this puzzle is beyond the reach of the clock semantics as it requires (in general) remembering the sequence of previous broadcasts of length proportional to the number of children. We discuss this further in §6.4.1.

3.9 Concluding remarks

We discuss an implementation of this theory based on symbolic techniques in Chapter 6.

Chapter 4

Synchronous digital circuits as functional programs

HARDWARE designs traverse a series of abstraction layers: what might begin as a high-level behavioural model that addresses architecture issues will, when mature, typically be manually translated into a *register-transfer level* (RTL) description that captures how the high-level computations are performed by finite-state means using logic gates and memories. This is typically validated against the original model using simulation and testing, or more formally with model checking techniques or a proof assistant. The resulting *net lists* (circuit schematics represented as graphs) are semi-automatically mapped to an implementation technology and laid out for realisation in silicon.

The original motivation for developing domain-specific languages (DSLs) (Mernik, Heering, and Sloane 2005) for the upper reaches of this process was to harness the huge increases in transistor densities on silicon chips forecast by Moore's law (Mead and Conway 1980). It was hoped that productivity would rise with the abstraction level, allowing designs to be more reusable, scalable and correct. Traditional imperative programming languages were a poor fit as their implicit sequentiality conflicts with the intrinsic parallelism of hardware, and a global store is in tension with the ideal of placing computations physically near the relevant state (Nikhil 2011). For these reasons simulation languages – specifically Verilog (based on C syntax) and VHDL (Ada) – were pressed into service as general-purpose hardware description languages (HDLs).

Despite their widespread use in industry, neither of these languages has been completely adequate. Their semantics are complex and have resisted useful formalisation (Boulton, Gordon, Gordon, Harrison, Herbert, and Van Tassel 1992; Gordon 1995). Only subsets of these languages can be synthesised to hardware, and these subsets need not be treated coherently by different tools. Moreover they lack modern semantically well-founded abstractions such as algebraic data types, higher-order functions (HOFs), overloading, subtyping and so forth. We contend that this leads to unnecessarily obfuscated descriptions, and greatly reduces the benefits of formal verification as it must be postponed until semantically-clear objects have been produced, which are typically low-level net-lists. This decreases the effectiveness and increases the cost of such techniques, as the cost of rectifying flaws is a function of when they are found (Brooks Jr. 1995). In addition the high-level structure and intuitions must somehow be rediscovered in these lower-level artifacts.

In the face of these deficiencies, many people have investigated how circuits may be described as functional programs, with most treating the common special case of synchronous digital circuits. Such models abstract the propagation delays of the combinational logic but not the transitions between states; our simulations are *cycle accurate* with respect to their realisation in hardware, and we have a global *clock*. In contrast an *asynchronous* model allows different components in a system to proceed independently (Jantsch and Sander 2005).

The majority of the methods we examine are *structural* techniques for combining system elements. These elements have *behavioural* descriptions and may represent subsystems at any level of abstraction; we do not require that they be synthesisable, though in our examples we will take them to be familiar logic gates. We will not go below the gate level as our synchrony assumption breaks down and resistive and capacitive effects begin to intrude (Axelsson, Claessen, and Sheeran 2005; Hanna 2000; Kloos 1987; Winskel 1986). We take advantages of a compositional semantics to be self-evident.

As implementers we would like to minimise the effort involved in providing the ever-increasing set of abstractions that users might like. One approach is to *embed* a DSL (to create an EDSL) into a suitably expressive meta language (Hudak 1996; Landin 1966), which allows the reuse of parsers, type checkers, optimisers, and some analysis tools while avoiding at least some of the myriad pitfalls of language design. We adopt Haskell syntax, with an idealised semantics, as an exemplar of the modern functional programming languages (Hughes 1989; Peyton Jones 2003) that have been shown to be attractive hosts.

Here we focus on the successful tradition of rendering synchronous digital circuits and similar systems as more-or-less pure first-order functional programs. The key features of this approach are the non-standard evaluation order and the use of higher-order functions to structure the descriptions, which we discuss at length in later sections. We concentrate in particular on the simulation semantics given to these circuits, and touch on other interpretations such as circuit layout, energy consumption, hazard detection, worst-case timing analysis and technology mapping; Sheeran (2005) explores these topics in more depth along one of the lines of research reviewed here. The pragmatics of these description mechanisms are just as important as the clarity of the semantics: there is little point in algebraic simplicity if the descriptions are too inconvenient to write and maintain.

We begin our survey by discussing a folklore rendition of synchronous digital circuits in a non-strict functional programming language before examining the hardware description projects that have used these techniques. Afterwards we consider some closely related subjects and topics of future research.

```
data Signal α = α :> Signal α

head :: Signal α → α
head (x :> xs) = x

tail :: Signal α → Signal α
tail (x :> xs) = xs

repeat :: α → Signal α
repeat x = x :> repeat x

map :: (α → β)
       → Signal α → Signal β
map f xs = f (head xs) :> map f (tail xs)

zip :: (α → β → δ)
     → Signal α → Signal β → Signal δ
zip f xs ys =
    f (head xs) (head ys) :> zip f (tail xs) (tail ys)

false, true :: Signal Bool
false = repeat False
true = repeat True

neg :: Signal Bool → Signal Bool
neg sig = map not sig

and2 :: Signal Bool → Signal Bool
          → Signal Bool
and2 sig1 sig2 = zip (&&) sig1 sig2

delay :: α → Signal α → Signal α
delay x sig = x :> sig
```

Figure 4.1: A simple embedded DSL for describing synchronous circuits.

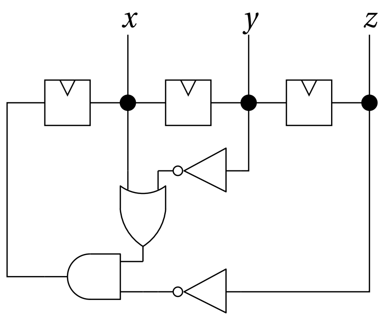
4.1 Circuit Semantics

One may expect the semantics of gate-level descriptions of synchronous digital circuits to be straightforward, and indeed the prevailing attitude amongst existing hardware description languages seems to be that lifting standard propositional logic to a temporal domain suffices for simulation (Camilleri, Gordon, and Melham 1986; Erkök 2002; Johnson 1983; O'Donnell 1987). For concreteness we capture the essence of this approach in the set of combinators shown in Figure 4.1, expressed in Haskell.

Here the non-strictness of our host language is crucial; the Signal α datatype models an infinite stream of values of type α. A proper value of this type has the form x_0 :> ... :> x_i :> ... for values x_i of type α, where the subscript indexes progression on an unbounded discrete timescale. Conversely it may be desirable for Signal α to be strict in the values it carries (of type α), to mitigate space leaks.

Circuits are first-order stream transformers of type Signal $\alpha \rightarrow$ Signal β, mapping streams of inputs to streams of outputs. State in sequential circuits is provided by a finite collection of initialised delay elements (clocked D-type flip flops) that provide access to values from the previous instant, and the instantaneous value of any wire is a function of the inputs and the values of the delay elements for that instant. This approach supports many useful equational laws that are often easier to apply than those for reasoning about arbitrary mutable state.

Our implementation of combinational logic is a pointwise lifting of the instantaneous operations to the temporal domain. As we use Haskell's recursion to model feedback, "cons" (our :> operator) should not evaluate its arguments (Friedman and Wise 1976). In other words evaluation is driven by data dependencies only.




```
trc :: (Signal Bool, Signal Bool, Signal Bool)
trc = (x, y, z)
  where
    x = delay False (and2 (or2 x (neg y))
                          (neg z))
    y = delay False x
    z = delay False y
```

Figure 4.2: A twisted ring counter as a set of first-order recursion equations.

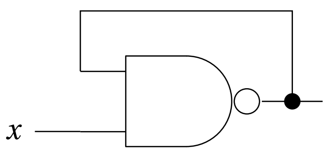
Clearly we can derive other operations such as xor, and as we will explore later in more detail, write succinct circuit generators in Haskell.

By way of an example, consider the twisted ring counter of Stavridou (1993, §3.3.2) shown in Figure 4.2. This circuit cycles through the sequence 000 → 100 → 110 → 111 → 011 → 001, which intuitively involves complementing the rightmost bit and moving it to the leftmost position, shuffling the others to the right. It *self stabilises*: whatever the state of the delay elements, the circuit will return to this sequence in a finite number of steps. In our description, each binding defines a wire, and the meaning of the whole network is the least fixed point of this set of equations. As such it is a *Kahn network* 1974; Claessen (2001, Chapter 5) presents many examples written in this style.

Note that each syntactic use of a gate in the description is intended to correspond to an actual gate in the hardware realisation. We will see that this expectation is in tension with the semantics of the host language in the same way that assuming that each procedure definition in a program is represented in the compiled object code is sometimes erroneous.

This encoding is termed a *shallow embedding* as there is no syntactic representation of circuits that can be manipulated from within Haskell. Its strength is that we can easily add new types of circuit elements, and freely reuse Haskell as a metalanguage. Its weakness is that we cannot manipulate descriptions from within the language, or reason about them inductively. In contrast a *deep embedding* would explicitly represent syntax, which can be challenging to define and use in a typed setting. Later we will see that Haskell's type classes provide a third way.

Our naïve semantics has an infelicity, however. Consider the following circuit:




```
f x = out
  where
    out = neg (and2 out x)
```

We can see that $f\,x$ diverges for all x by considering the definition of and2 in Figure 4.1 and the semantics of (&&) shown in Figure 4.3.

(&&)	⊥	F	T
⊥	⊥	⊥	⊥
F	F	F	F
T	⊥	F	T

and	⊥	F	T
⊥	⊥	⊥	⊥
F	⊥	F	F
T	⊥	F	T

pand	⊥	F	T
⊥	⊥	F	⊥
F	F	F	F
T	⊥	F	T

Figure 4.3: The truth tables of short-circuit and (&&) standard to most programming languages, bi-strict and and parallel and. Values for the first argument are on the left, and for the second on the top. The value $\perp$ denotes a diverging argument.

In contrast the symmetric variant $f'\ x\ =\ out$ **where** $out\ =\ \mathsf{neg}\ (\mathsf{and2}\ x\ out)$ converges to true on the argument false. This behaviour is termed *short-circuit evaluation* in strict languages such as ML and C.

As f and f' have isomorphic circuit diagrams, we expect them to have the same semantics, and therefore and2 should make symmetric use of its inputs. One option is to make and2 strict in the head of its second argument, causing both f and f' to always diverge. This yields the traditional model where every well-defined loop is required to contain a delay, and as we will see, this must be the semantics intended by the champions of the approach sketched above. Here we explore the less trodden path of making and2 non-strict in both its arguments.

To motivate this choice, consider the classic example due to Malik (1993) shown in Figure 4.4. For any circuits f and g, this circuit generator is intended to dynamically choose between $f \circ g$ and $g \circ f$ using only single copies of f and g and three multiplexers. (A multiplexer chooses between one of its inputs on the basis of an auxiliary input.) What makes this design work is that the apparent combinational cycles in the schematic cannot be realised dynamically, i.e., every assignment to the inputs yields an acyclic path through the circuit, assuming that the multiplexers are symmetrically non-strict in the inputs they choose between. If we construct such multiplexers from the basic gates and2 and neg, then and2 must be lazy in at least one argument for this to obtain. We discuss this example further in §4.2.3.

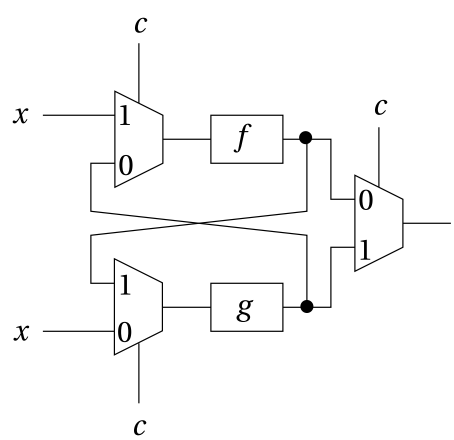


Figure 4.4: A useful cyclic circuit schema that, for arbitrary f and g, computes either $(f\ \circ\ g)\ x$ or $(g\ \circ\ f)\ x$ depending on the input c. Without cycles two copies of f and g would be required.

Another example is the hardware bus arbiter of R. de Simone that is naturally rendered as a combinationally-cyclic circuit (Potop-Butucaru, Edwards, and Berry 2007, §2.3). Fairness is enforced by circulating a token around a ring of arbiter cells, and the token holder can delegate permission to proceed to the succeeding cells in the ring.

These cycles also arise naturally when we abstract from the gate to the functional block level, as observed by Burch, Dill, Wolf, and De Micheli (1993). Their example of a carry-lookahead adder requires the adders and carry-lookahead generator to instantaneously interact across an abstraction boundary.

Semantically we can treat cyclic combinational logic in the same way as other recursive definitions, by using a *domain* (Winskel 1993); in this instance we introduce a third value to our Bool type and impose a (partial) *information ordering* on these values:

$$\begin{tikzcd}[column sep=small] \mathrm{F} & & \mathrm{T} \\ & \bot \arrow[ul, no head] \arrow[ur, no head] & \end{tikzcd}$$

This is to say that the undefined value $\bot$ is less defined than either of T and F, and that these two proper values are distinct. Intuitively we take $\bot$ to mean that the wire does not settle to a valid value, with F and T representing the standard stable Boolean values. We emphasise that $\bot$ is not so much an unknown value as an invalid one in this semantics.

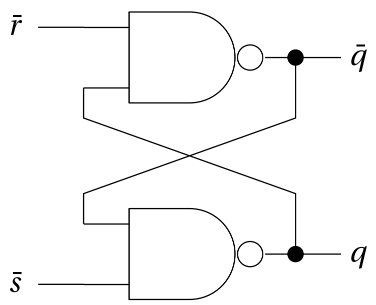
Using this domain we can give a symmetrically non-strict semantics to our and2 primitive using the pand function shown in Figure 4.3, which also shows two of its stricter cousins for comparison. With unfortunate consequences for our simple embedded DSL of Figure 4.1, Plotkin (1977) showed that pand is not *sequentially* computable; see Gunter (1992, §6.1) and Brookes (1993) for further background on this point. As most functional languages are intended to have such a deterministic sequential semantics, we should use the stricter and operation if we rely on the host language's recursion. If we wish to support combinational cycles then we need to adopt an alternative semantics for recursion, such as explicit iteration of some reified representation, which implies that we can no longer write our circuits as simple recursion equations in the host language. Alternatively one could introduce support for parallel or non-deterministic operations into the host language, but doing so can severely complicate its implementation and semantic properties (Hughes 1983; Moran 1998).

We note that the sequential behaviour of circuits is unaffected by this change to the combinational semantics; we continue to use non-strict sequences. However this may not be true if we wish to accommodate non-determinism, or ways of observing the circuit without changing its interface, such as for debugging purposes. How invasive this is could be taken as a measure of how flexible the methods of the following sections are.

Constructive circuits always assign all wires non-$\bot$ values when always fed non-$\bot$ inputs; these can be unfolded into semantically-equivalent acyclic circuits, which can then be passed to tools that do not directly support combinational loops. These circuits are termed "constructive" due to their relationship with intuitionistic propositional logic. Such circuits have been used to give a semantics to an imperative synchronous language (see §4.3.1).

Combinational cycles trade time for space, and convergence may require time exponential in circuit size (Shiple, Berry, and Touati 1996) in the presence of nested loops. Neiroukh, Edwards, and Song (2008) found references to these types of circuits stretching back to switching theory in the 1960s. Shiple et al. (1996) have grounded this parallel semantics in the physical models of Brzozowski and Seger (1995). The connection with constructive logic continues to be explored by Mendler, Shiple, and Berry (2012), and Riedel and Bruck (2003) show that cycles can yield significant space reductions in practice.

The reader should not be seduced into believing this semantics completely reflects the physical behaviour of cyclic circuits. Consider the classic set-reset latch:



$$
\begin{aligned}
&\mathit{srLatch}\ \bar{s}\ \bar{r} = (q, \bar{q}) \\
&\quad \textbf{where} \\
&\qquad q = \mathsf{neg}\ (\mathsf{and2}\ \bar{s}\ \bar{q}) \\
&\qquad \bar{q} = \mathsf{neg}\ (\mathsf{and2}\ \bar{r}\ q)
\end{aligned}
$$

While the structural description on the right is accurate, the semantics we have ascribed to the primitives does not yield the desired latching behaviour as observed in practice. This is because the retention of the latch's value across cycles depends crucially on the propagation delays that our assumption of synchrony has already abstracted from, and the semantics presented here does not retain the values of wires between cycles. Similarly tri-state busses may not be properly treated by this semantics either.

Descriptions in this style are quite pleasant as the connection with the circuit's net list is quite clear, and there is no extraneous sequentiality; these recursion equations encode data dependency amongst the components and nothing more. Moreover we can easily incorporate subsystems described at more abstract levels than primitive gates for the purposes of high-level design validation. However giving these descriptions alternative semantics, such as an explicit representation of a circuit's net list, is difficult in a pure host language. We discuss this issue in §4.2.3 and later sections.

4.2 Circuits and Functional Programming

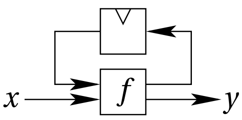
Having sketched the semantics we might expect of an HDL for synchronous digital circuits, we now review systems that represent circuits using functional programming languages. We begin with the the combinatory approach of μFP, and the contemporaneous use of recursion equations by Johnson. Hydra bridges the two traditions and points the way to the Haskell-hosted

Lava systems that continue to be developed. We discuss the Hawk project that applied these techniques to microarchectures, the Jazz system, and the Cryptol® language for describing implementations of cryptographic primitives. We conclude with some higher-level behavioural techniques.

4.2.1 μFP

Sheeran (1984) based her μFP system on the FP language of Backus (1978), who championed a combinatory style of programming now termed *point-free*. In essence, function composition is emphasised over application, and algebraic laws are prized (Bird 1987; Meijer, Fokkinga, and Paterson 1991).

μFP extends FP by lifting instantaneous operations to streams with the α combinator, better known as map, and a delay operator μ:

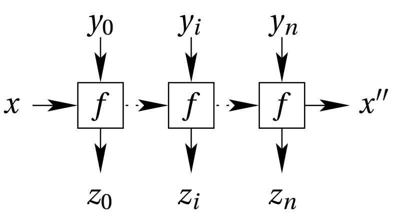


$$\begin{aligned} &\mu :: ((\mathsf{Signal}\ \alpha,\ \mathsf{Signal}\ \delta) \rightsquigarrow (\mathsf{Signal}\ \beta,\ \mathsf{Signal}\ \delta)) \\ &\quad \to \mathsf{Signal}\ \alpha \rightsquigarrow \mathsf{Signal}\ \beta \\ &\mu\, f \ = \ \lambda x.\ \mathbf{let}\ (y,\ z) \ = \ f\ (x,\ \mathsf{delay}\ ?\ z)\ \mathbf{in}\ y \end{aligned}$$

The diagram on the left depicts μf for an arbitrary circuit f, and on the right is a simulation semantics for μ in Haskell. The latter should not be taken too literally as both FP and μFP are untyped, and the only constraints on the implementations of combinators is that they satisfy the associated laws. We again informally identify the type of wires with the Signal domain. A strength of the combinatory approach is that the type of circuits $\alpha \rightsquigarrow \beta$ which map inputs of type α to outputs of type β can be separated from the function space of the meta language $\alpha \to \beta$. Note that the register introduced by μ is initialised by the "don't care" constant ? value.

Circuits are described structurally and given two semantics: simulation, by translation into the sequence type of FP along the lines of what we sketched in §4.1, and layout using the DSL for functional geometry of Henderson (1982). This early example of reinterpretation was realised as a custom processor rather than an embedding in a host language.

Higher-order functions (HOFs) capture the regularity of data-oriented circuits in an elegant manner. For example, the row combinator[1] expresses a common pattern used, for instance, in a simple ripple-carry adder:



$$\begin{aligned} &\mathsf{row} :: ((\alpha,\ \beta) \rightsquigarrow (\alpha,\ \delta)) \\ &\quad \to (\alpha,\ [\beta]) \rightsquigarrow (\alpha,\ [\delta]) \\ &\mathsf{row}\ f\ (x,\ [\,])\ =\ (x,\ [\,]) \\ &\mathsf{row}\ f\ (x,\ y : ys)\ = \\ &\quad \mathbf{let}\ (x',\ z)\ =\ f\ (x,\ y) \\ &\quad\quad (x'',\ zs)\ =\ \mathsf{row}\ f\ (x',\ ys) \\ &\quad \mathbf{in}\ (x'',\ z : zs) \end{aligned}$$

[1] The row function is called mapAccumL in the standard Haskell Data.List module.

We note that such structural definitions are much more intuitive and less verbose than a typical generic definition in VHDL, where the use of array indices introduce the spurious possibilities of off-by-one errors and so forth.

μFP emphasises composition and not the primitive circuits; the latter are not further specified by Sheeran (1984). Instead a fixed set of higher-order combining forms that have good geometric and algebraic properties are studied. Sheeran observes that almost all the laws of FP apply to μFP, with the notable exception of a conditional distribution law. The FP version is as follows:

$$h \circ (i \longrightarrow t\,;\, e) = (i \longrightarrow h \circ t\,;\, h \circ e)$$

where

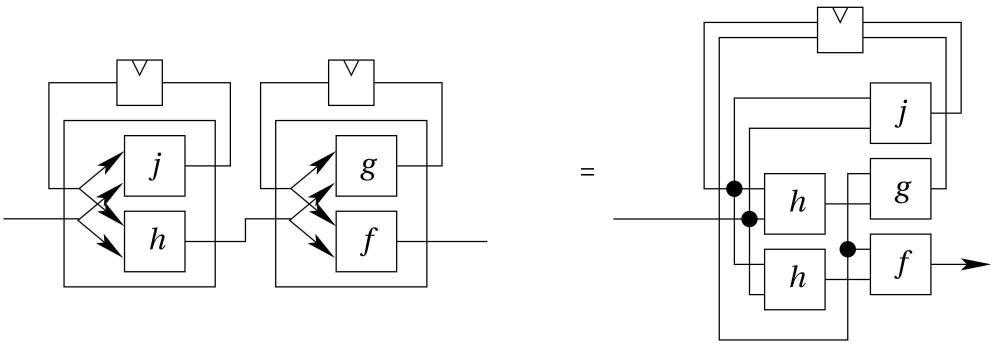
$$\begin{aligned}(- \longrightarrow -\,;\,-) &:: (\alpha \to \mathsf{Bool}) \to (\alpha \to \beta) \to (\alpha \to \beta) \to \alpha \to \beta \\ (i \longrightarrow t\,;\, e) &= \lambda\alpha.\,\textbf{if}\ i\,\alpha\ \textbf{then}\ t\,\alpha\ \textbf{else}\ e\,\alpha\end{aligned}$$

Lifting $(- \longrightarrow -\,;\,-)$ to μFP is an exercise in tuple spaghetti:

$$(i \longrightarrow t\,;\, e)_{\mu\text{FP}} = \mathsf{map}\ (\pi^3_1 \to \pi^3_2\,;\,\pi^3_3) \circ \mathsf{zip3} \circ [i,\ t,\ e]$$

where $[i,\ t,\ e]$ is informal notation for the fanout $\lambda x.\ (i\,x,\ t\,x,\ e\,x)$ and π^n_i projects the ith component of an n-tuple. In μFP, h must be combinational for the putative equation to hold, for there is always a stateful h that can distinguish t from e if they are different.

Sheeran also proposed a fixed-point fusion rule for her μ construct:



$$\begin{aligned}\mu[f,\ g] \circ \mu[h,\ j] = \mu[&f \circ [h \circ [\pi^2_1,\ \pi^2_2 \circ \pi^2_2],\ \pi^2_1 \circ \pi^2_2], \\ &[g \circ [h \circ [\pi^2_1,\ \pi^2_2 \circ \pi^2_2],\ \pi^2_1 \circ \pi^2_2],\ j \circ [\pi^2_1,\ \pi^2_2 \circ \pi^2_2]]]\end{aligned}$$

This law is intended to be used as a *fission* law, in the right-to-left direction: it moves the independent parts of a state-holding element closer to the relevant computations. We note that this law does not hold in our $\mathsf{Signal}\ \alpha$ domain due to the presence of partial lists; we discuss this issue further in §4.3.5.

A major source of discomfort in the purely combinatory style of programming is the need to explicitly route values from definition to use; in the applicative style we used in §4.1 the λ-calculus provides this service implicitly by allowing us to give names to wires in some scope.

This *plumbing problem* is certainly why raw combinators are generally thought of as compilation targets and not source languages.

μFP has been applied to the design of circuits with regular structure such as adders, multipliers and a correlator, and more generally to systolic arrays, where critical path lengths are reduced by pipelining in a hazard-free non-recursive way. The process begins with purely combinational circuit designs which are transformed into sequential pipelines by retiming transformations. Through a disciplined use of the state introduced in this final step, the original and retimed circuits can be very simply related. All examples are data-oriented, and control-oriented circuits do not tend to have the geometric regularities that these combinators capture.

Sheeran (2005) reviews this line of research as well as her work on some of the descendants of this system that we discuss later in this article.

4.2.2 Hardware synthesis from first-order recursion equations

Johnson and his collaborators have made an extended investigation into the practical use of derivational reasoning in digital design (Johnson 1983, 2001; Johnson and Bose 1997). Their goal is to provide tools to explore the space of implementations of a high-level behavioural specification. Here synchronous digital circuits are represented as first-order recursion equations over streams as we discussed in §4.1.

The first major application of these techniques (Johnson 1983, Chapter 5) was to the refinement of an interpreter for a higher-order language into a stack-based virtual machine using the approach developed by Wand (1982). This process relied on general results about flowchart schemata (Greibach 1975; Manna 1974), such as the fact that all tail-recursive functions can be implemented in constant space, and arbitrary functions can be evaluated using a stack. Given that these schemata can be captured by higher-order functions, we can see this as a control-oriented complement to the μFP agenda, but where the original specifications are behavioural and more abstract.

This process uses the program transformation framework of Burstall and Darlington (1977), with the preservation of total correctness left to the discretion of the designer (Johnson 1983, §2.4.5). Circuits represented as recursive streams are optimised using equations similar to those in the previous section (Johnson 1983, Chapter 6). Suitably oriented, these equations can transform circuits that operate on their arguments in parallel into sequential ones.

An untyped lazy functional programming system by the name of Daisy was the vehicle for this research, and circuit descriptions were manipulated by hand. A strength of this approach is that all refinement artifacts are executable, i.e., can be experimented with programmactically.

Building on this work, Johnson and Bose (1997) developed the DDD tool. Here the refinement process begins with a first-order specification expressed as a pure iterative (tail recursive) function in the strict untyped functional language Scheme, extended with a mechanism for recursively defining streams. These are structurally decomposed into putative hardware blocks, again using the Burstall and Darlington rules. From these DDD mechanically generates architectural descriptions consisting of control and datapath circuitry, which are further optimised using laws about recursive stream transformers like those we have seen before. Finally representations of abstract types such as numbers are chosen and shown sufficient using data refinement.

Some of these steps have side conditions, such as showing that a fixed-width binary representation of a number is adequate. Paul Miner (Johnson 2001, §3.3) experimented with using the PVS proof assistant to demonstrate these conditions and the soundness of circuit optimisations but was stymied by the lack of support for infinite streams in proof assistants at the time.

This approach was used to derive implementations of the FM8501 and FM9001 processors due to Hunt (Bose and Johnson 1993), a PCI bus interface and a Java byte code generation core; it is claimed to be a useful technique for circuits with high algorithmic complexity. Similarly to μFP, it does not address the specification of interfaces, power supplies or clock trees.

4.2.3 Hydra

Hydra (O'Donnell 1987, 1992, 1995, 2003) is a long-running experiment in representing circuits in various pure functional programming languages following Johnson's tradition of circuit transformation. It holds fast to the idea of directly expressing circuits in the host programming language and reasoning equationally in that language. This has the advantage over μFP of allowing new combining forms to be introduced by the end user of the system.

The central problem with this approach is of identifying shared subcircuits. Consider this rendition of the fgORgf circuit in Figure 4.4 in the style of Hydra (and Lava 2000 which we meet later in this article):

$$
\begin{aligned}
&\mathsf{fgORgf} :: (\mathsf{Signal}\ \alpha \rightsquigarrow \mathsf{Signal}\ \alpha) \to (\mathsf{Signal}\ \alpha \rightsquigarrow \mathsf{Signal}\ \alpha) \\
&\qquad\quad \to (\mathsf{Signal\ Bool}, \mathsf{Signal}\ \alpha) \rightsquigarrow \mathsf{Signal}\ \alpha \\
&\mathsf{fgORgf}\ f\ g\ (c, x) = out \\
&\quad \textbf{where} \\
&\qquad fOut = f\ (\mathsf{mux}\ (c, x, gOut)) \\
&\qquad gOut = g\ (\mathsf{mux}\ (c, fOut, x)) \\
&\qquad out = \mathsf{mux}\ (c, gOut, fOut)
\end{aligned}
$$

where the mux combinator is defined as:

$$
\begin{aligned}
&\mathsf{mux} :: (\mathsf{Signal\ Bool}, \mathsf{Signal}\ \alpha, \mathsf{Signal}\ \alpha) \rightsquigarrow \mathsf{Signal}\ \alpha \\
&\mathsf{mux}\ (c :> cs,\ x :> xs,\ y :> ys) = (\textbf{if}\ c\ \textbf{then}\ x\ \textbf{else}\ y) :> \mathsf{mux}\ cs\ xs\ ys
\end{aligned}
$$

If we think of fgORgf as a standard Haskell definition then we can apply the unrestricted β-rule to unfold the definition of *gOut* in the definition of *fOut*:

$$fOut = f\ (\mathsf{mux}\ c\ x\ \underbrace{(g\ (\mathsf{mux}\ c\ fOut\ x))}_{gOut})$$

This new circuit is extensionally equal to the previous one, and so these should not be distinguished by any Haskell context. However they are clearly structurally distinct, as the new version uses two copies of f. In other words, β-reduction invalidates our identification of function definitions with hardware gates. Therefore we seek a way to make these circuits observably different while retaining enough of the host language's semantics to support the kind of equational reasoning that circuit transformations depend upon.

O'Donnell has proposed several ways of resolving this reification problem (see also Claessen (2001, Chapter 3)):

- In more pragmatic times, O'Donnell (1987) suggested the use of pointer equality to reify the expression graph of the circuit. This is a non-conservative extension to a pure language, rendering the foundational β-rule potentially unsound everywhere, thereby destroying equational reasoning.

- O'Donnell (1992) asked the circuit designer to do what the language processor could not; a combinator is added so that labels can be manually attached to components. This approach is inconvenient, non-compositional and impedes the use of higher-order combinators such as row.

- Most recently, O'Donnell (2003) advocated the manipulation of the circuits as Haskell abstract syntax using Template Haskell (Sheard and Peyton Jones 2002). This is at best a partial solution as the syntax for circuits and generators are not clearly separated here; intuitively we expect to run a circuit generator, perhaps using higher-order combinators as canvassed in §4.2.1, that yields the abstract syntax of a particular circuit. As the generators are arbitrary definitions in a Turing-complete language, it is difficult to see how this approach is any easier than writing a traditional standalone language processor.

  In any case manipulating the abstract syntax of the host language is fraught with semantic issues and runs the risk of destroying many of the reasoning principles valued by functional programmers. We discuss this approach further in §4.4.

Hydra supports a variety of circuit semantics (O'Donnell 1995), though as we observed earlier, below the synchronous gate level lurk many subtle issues. O'Donnell and Rünger (2004) designed a carry lookahead adder using Hydra as a notation for reasoning in the Squiggol style popularised by Bird (1987).

4.2.4 Lava

The original Lava system (Bjesse, Claessen, Sheeran, and Singh 1998) was an attempt to embed a flexible hardware description language into pure Haskell in such a way that circuit descriptions could be both generated and manipulated within the host language. *Type classes* (Kaes 1988; Wadler and Blott 1989) were used to give a signature for the circuit primitives. By parametrising these with a *monad* (Wadler 1997), each interpretation of a circuit in Lava could encapsulate the effects it requires. For instance, a net list interpretation may use a state monad to assign a number to each wire and map each basic component into a graph node. Effects such as non-determinism or probing encapsulated signals can be easily modelled using appropriate monads. This is the middle path between shallow and deep embeddings mentioned in §4.1, and is now termed a *finally tagless* representation (Carette et al. 2009).

The provided loop combinator supports cycles in sequential logic:

$$\mathsf{loop} :: \mathsf{CircuitMonad}\ m \Rightarrow (\alpha \to m\,\alpha) \to m\,\alpha$$

where the CircuitMonad class is the signature of this and the other basic circuit combinators. Intuitively such a recursion operator should perform the effects of its argument computation only once while providing the computation access to the value it finally yields. This invalidates an unfolding semantics, and therefore the application of the β-rule that duplicated circuitry in §4.2.3, while preserving this law in the purely functional parts of the language. Erkök (2002) later gave an axiomatic treatment of these operators, and developed a syntax to reduce the syntactic burden when defining several values by simultaneous monadic recursion. Here is our fgORgf example is this style[2]:

```
fgORgf :: CircuitMonad m
        ⇒ (α → m α) → (α → m α) → (Bool, α) → m α
fgORgf f g (c, x) =
  do rec fOut ← mux (c, x, gOut) >>= f
         gOut ← mux (c, fOut, x) >>= g
     mux (c, gOut, fOut)
```

where the bind operator ($\gg\!=$) is a monadic equivalent to (reverse) function application, and mux now has type $\mathsf{CircuitMonad}\ m \Rightarrow (\mathsf{Bool}, \alpha, \alpha) \to m\,\alpha$; our type of circuits $\alpha \rightsquigarrow \beta$ is concretely $\alpha \to m\,\beta$.

We contend that this description is almost as syntactically appealing as those in Hydra (§4.2.3) and Lava 2000 (§4.2.5). However the monadic structure makes visible the order in which the components of the circuit are defined (Claessen 2001, §1.8); in other words, a circuit can be given two semantically distinguishable descriptions in this notation simply by permuting the monadic commands. We might attempt to repair this infelicity by requiring that our monad be commutative, i.e., that it is insensitive to such permutations, but clearly any interpretation that assigns unique names to the gates will fail to have this property. This lack of full abstraction also complicates formally reasoning about circuit equivalences.

The original Lava system suffered somewhat from the limitations of using single-parameter type classes for reinterpretation, and successor systems such as Hawk (§4.2.7) experimented with generalisations.

[2]The **do rec** syntactic form has displaced the keyword **mdo** introduced by (Erkök 2002).

4.2.5 Lava 2000

Lava was later refined by Claessen (2001) into the Lava 2000 system, which is an embedded DSL for parametrised circuits whose instances are fed into myriad tools for analysis: simulation and realisation in hardware via the industry-standard VHDL, model checking of various kinds (Clarke et al. 1999; Halbwachs, Lagnier, and Raymond 1993), testing with QuickCheck (Claessen 2001, Chapter 4) and so forth. This design-and-verify approach contrasts sharply with the transformational correct-by-construction approaches championed by Sheeran (§4.2.1), Johnson (§4.2.2) and O'Donnell (§4.2.3), all of which rely on equational reasoning in the host language.

In Lava 2000 *circuit generators* are standard Haskell expressions as we saw in §4.2.3. When run, these expressions generate a description of a concrete circuit which is reified into a data structure by disciplined pointer-equality testing. This is termed *observable sharing*. In contrast to the earlier systems Lava 2000 has no need of a precise semantics for its host language as it is merely the language of circuit generators, which are only executed and not analysed.

Claessen (2001, §3.3.4) notes that observable sharing makes visible the difference between call-by-need (laziness) and call-by-name (non-strictness): circuits without parameters are shared whereas those with parameters are duplicated, acting like templates. This loss of the β-rule of the λ-calculus is hardly surprising – we are trying to identify sharing, which is precisely the distinction between these semantics. At the source level this problem is ameliorated by the adoption of a particular style of description that is less likely to trap the unwary. It also relies on defeating compiler optimisations such as common-subexpression elimination and the full laziness transformation (Peyton Jones 1987) that introduce sharing.

Lava 2000 additionally marked a departure from using the underlying lazy functional programming language to give a direct semantics for circuits: instead, the circuit generator builds a monomorphic graph describing the final circuit, which is then interpreted by traversal. Extra types of circuit elements such as non-deterministic choice can be modelled as distinct kinds of graph nodes. Circuits are therefore a subset of Haskell expressions that are treated as abstract syntax, similarly to O'Donnell (2003) but within a single metalanguage.

This approach allows Claessen (2003) to handle circuits with combinational cycles by computing explicit (reified) fixed points, but precludes the possibility of polymorphic signals: circuits in Lava talk about bits and integers only. Moreover it limits the possibility of transmitting some of the structure of the circuit generator to the backends without extensive surgery to Lava 2000 itself. For instance, it may be more efficient for a tool consuming these descriptions to generate a single instance of a circuit and copy that as required instead of receiving the entire description of a subsystem at each point of use. Also by allowing arbitrary HOFs as combining forms, circuits in Lava 2000 do not always have reasonable layouts.

Lava 2000 and a variant designed by Satnam Singh at Xilinx (§4.2.6) were applied to the design and realisation of a sorter core based on Batcher's butterfly techniques (Claessen, Sheeran, and Singh 2003). They have also been used to analyze many other combinational circuits such as adders and multipliers (Axelsson 2003), and as a host for a sequential language much simpler than what we discuss in §4.3.1 (Claessen 2001, Chapter 6). More recently Sheeran 2005; 2011 has developed techniques for context-sensitive circuit generators and optimisers using this system.

4.2.6 Other Lavas

"Lava" has come to denote the structural description of hardware in Haskell. We briefly review three of these systems.

Xilinx Lava

As mentioned earlier, Singh developed a variant of Lava while at Xilinx, Inc. as an experimental vehicle for mapping circuits to the company's Virtex line of Field Programmable Gate Arrays (FPGAs, a type of reconfigurable hardware). In contrast to other Lavas, this system included explicit layout combinators similar to those in μFP (§4.2.1) (Singh and James-Roxby 2001). Singh (2011) shows that user-specified layouts remain useful in some cases.

Circuit descriptions are similar to those in Lava 2000. Primitive gates are specified in terms of the look-up tables that FPGAs provide. Sharing is accounted for using a monad internally, which creates a monomorphic graph that is then translated into VHDL (etc.) for consumption by external tools. There is no support for cycles of any kind.

In addition to the sorter network mentioned above, Xilinx Lava was used to describe dynamic (runtime) reconfiguration and specialisation (Singh 2004). Unusually for a Lava, clock signals are explicitly mentioned in descriptions, allowing a stateful circuit to be suspended through clock gating.

York Lava

Naylor and Runciman (2012) use York Lava to describe their Reduceron graph-reduction processor, which runs on an FPGA. This is a revival of the idea of programming-language specific processors that avoid the von Neumann bottleneck of a single global store. Such experiments are far easier to carry out now as reconfigurable hardware is quite affordable, and more likely to be adopted as the sequential performance of standard processors flatlines. The source language is the kernel of a lazy language compatible with Haskell. The processor is described in a mix of recursion equations and an imperative behavioural language they call Recipe, which is given a semantics by translation into their Lava.

The semantics of York Lava is standard. The project investigated the use of explicit *fork points* to signal sharing (Naylor and Runciman 2009): the overloaded `fork` combinator should be used to indicate that a wire has multiple sinks. This allows most useful circuits to be reified while retaining the purity of the host language in a manner ultimately quite similar to the explicit use of recursion combinators. However this approach was abandoned in favour of Lava 2000-style pointer comparisons.

Layout is performed by the FPGA toolset.

Kansas Lava

The Kansas Lava system is a vehicle for investigating circuit transformation and refinement. Gill and Farmer (2011) report on the "semi-formal" derivation of an error-correcting code using the worker/wrapper transformation (Gammie 2011c; Gill and Hutton 2009), in concert with applicative functors (McBride and Paterson 2008) and type functions (Chakravarty, Keller, Peyton Jones, and Marlow 2005). In contrast to the structural use of lists we saw in §4.2.1, the dimensions of vectors and matrices are encoded in their types, which is both safer and more awkward as present Haskell systems do not have full support for type-level arithmetic. Layout is not prescribed.

Gill (2009) previously advocated another solution to the reification problem: instead of polluting the semantics of the pure core of Haskell by making the sharing of values observable at all types ala Lava 2000 (§4.2.5), scrutinising the structure of a circuit is confined to the IO monad, where anything goes. Once again a test for pointer equality is employed, and this extra discipline makes the approach both safer – one is less likely to accidentally exploit the observation of sharing – and more obscure, as the semantics of the IO monad is complex, fluid and yet to be formally specified. Moreover it suffers from exactly the same problem as Lava 2000: by allowing call-by-name and call-by-need semantics to be distinguished, the β-law of the λ-calculus fails, as we previously remarked. This may complicate relating fully-formal derivations and Kansas Lava circuits and generators.

This system uses the standard Kahn network semantics for circuits, and maintains both a shallow and deep embedding of the circuit to allow for direct simulation and VHDL export. As a result the simulation semantics of the circuits is not isolated from Haskell's, which precludes a treatment of combinational cycles. Clock information is explicitly encoded into types in a manner similar to Lucid Synchrone (see §4.3.1).

Layout is performed by external tools.

4.2.7 Hawk

Hawk (Launchbury, Lewis, and Cook 1999; Matthews, Cook, and Launchbury 1998) is a DSL embedded in Haskell for describing and reasoning about microarchitecture. Semantically it is very traditional, employing non-strict lists of values to model synchronous systems, though it does not require nor guarantee that these systems be finite-state.

The emphasis of this system is on algebraic abstractions of pipelined microprocessor designs using transactions, which record the relevant state of the system for each instruction as it proceeds through the pipeline. This requires more type structure than allowed by Lava 2000. Early versions of Hawk attempted to use the type classes and monads of the original Lava, but this approach was abandoned due to the difficulty of finding a suitable recursion combinator, and the lack of methods for resolving ambiguous uses of multi-parameter type classes that represent relations between types. Many of the issues they identified were soon addressed (Chakravarty et al. 2005; Erkök 2002; Jones 2000). Later versions of Hawk provided only a simulation semantics along the lines of §4.1.

The proposed algebraic laws for manipulating microarchitectures were verified in Isabelle/HOL (Nipkow et al. 2002), for which the theory of *converging equivalence relations* was developed by Matthews (1999) to allow the definition of recursive functions in HOL over infinite sequences. Under mild conditions such functions have unique fixed points, and unlike the domain theoretic approach, uncomputable functions can be defined. We discuss formal models further in §4.3.5.

The Hawk group built models of the then state-of-the-art Intel Pentium Pro in addition to the DLX, a standard example of a pipelined processor. Matthews (2000) reviews the project and discusses how Hawk relates to other HDLs.

4.2.8 Cryptol®

Cryptol® is a proprietary DSL and toolset developed by Galois, Inc. for compiling descriptions of cryptographic algorithms into hardware or software (Browning and Weaver 2010). The language provides only bits as a primitive type, with sized sequences and tuple constructors used to aggregate values. Its type system is very flexible, allowing the definition of size- and type-polymorphic functions, and constraints allow sizes to be underspecified. Cryptol® descriptions can be checked for equivalence using external tools such as SAT and SMT solvers.

Combinational circuits are described applicatively, as in §4.1, but as instantaneous functions. These can be lifted to streams pointwise, as before, or as transition functions for state machines in the coiterative style using an `unfold` combinator. The language restricts the use of higher-order functions to those that can be unfolded at compile time, which is often sufficient for the sort of circuit combinators discussed in §4.2.1. Partial application is not supported, and functions are uncurried.

A construct similar to Haskell's list comprehensions is used to define streams recursively, which is realised as delayed feedback in the generated circuit. It is also used to traverse finite sequences, and the language goes beyond purely structural descriptions by providing `par`, `seq` and `reg` combinators that specify how the comprehension should be scheduled in time and space. Browning and Weaver (2010, §3.4) show that, by default, mapping a function f across a finite sequence s yields hardware with as many fs as the width of s, whereas the `seq` annotation generates only a single f and the requisite synchronous scheduling logic to process s sequentially. The `reg` combinator pipelines a circuit in a standard way.

Layout is once more performed by external tools.

4.2.9 Jazz

The Jazz system was developed by A. Frey, with contributions from F. Bourdoncle, G. Berry, P. Bertin and J. Vuillemin, contemporaneously with the original Lava system (Claessen 2001, §1.11). It had a Java-inspired syntax but was in fact a higher-order, lazy, purely-functional language that supported the combination of subtyping and parametric polymorphism proposed by Bourdoncle and Merz (1997). Novelly it provided native support for the arithmetic over 2-adic integers (streams of booleans) due to Vuillemin (1994). The elaboration of circuit descriptions into netlists is similar to Lava's approach. Descriptions can be given multiple interpretations by the standalone language processor.

4.2.10 High-level Hardware Synthesis

At a higher level we might hope to abstract from timing behaviour by compiling *behavioural* descriptions into synchronous or asynchronous circuits. Several such systems are based on ideas closely related to functional programming.

SAFL (Mycroft and Sharp 2003) is a first-order pure functional language with a strict semantics where the only program schema on offer is tail recursion. As each function in a SAFL description is mapped to a hardware block, the key task of its FLaSH compiler is to schedule the use of these blocks when they are called from multiple places in the source program.

A similar approach was taken in the design of the SASL first-order stream processing language (Frankau and Mycroft 2003). Tail-recursive functions define streams, where each iteration yields zero or more elements. Unlike Cryptol® (§4.2.8), functions can be defined by recursion over scalar (non-stream/vector) types. Static allocation is ensured by an affine type scheme that ensures streams are read at most once. In contrast to our model and that of the synchronous languages we discuss in §4.3.1, streams are not clocked: explicit handshaking is used to signal completion and demand more input. Under the typing constraints this allows arbitrary streams to be merged in finite space, whereas in the synchronous language Lustre the streams would need to be on the same clock.

The ongoing "geometry of synthesis" project of Ghica (Ghica 2007; Ghica, Smith, and Singh 2011) interprets a higher-order imperative language – a variant of Reynolds's Idealised Algol – into various kinds of logic. It relies on Reynolds's Syntactic Control of Interference as realised by an affine type system to eliminate conflicting writes to shared state. Unlike Johnson's approach (§4.2.2) it is fully automatic.

Bluespec (Arvind and Nikhil 2008; Nikhil 2011) schedules sets of guarded commands into time slots where the actions are executed transactionally. It began with a syntax close to Haskell's, with many of its structuring facilities, and has since adapted to the SystemVerilog and SystemC ecosystems while retaining many of its novel features.

4.2.11 Concluding remarks

The various Lavas solve the issue of identifying shared subcircuits in different ways; some use observable sharing, either by asking the user to explicitly name certain nodes in the graph (Hydra, §4.2.3), or implicitly (Hydra, §4.2.3, Lava 2000, §4.2.5 and Kansas Lava, §4.2.6). Others use monadic recursion (the original Lava, §4.2.4 and Xilinx Lava, §4.2.6). Another suggested marking fanout with explicit fork combinators (York Lava, §4.2.6). A *linear* variant of the *implicit parameters* of Lewis, Launchbury, Meijer, and Shields (2000) was also proposed but was later deemed to be too semantically complex in practice. We discuss a further alternative of more fully insulating the language of circuit generators from that of circuits in §4.4.

4.3 Related Work

Having reviewed the state-of-the-art in describing digital synchronous circuits as functional programs, we briefly discuss some areas that lie alongside ours: we point into the voluminous literature on synchronous programming languages and algebraic techniques for hardware description, consider the role of relational models, and sketch some of the issues with formal functional models.

4.3.1 Synchronous Languages

The synchronous programming languages have deployed similar ideas to those of sequential digital circuits to achieve *deterministic concurrency* in software, and *reactive systems* more generally. Berry (1999b) argues forcefully for determinacy:

> Nondeterministic systems are harder to specify, and it is not even trivial to define a good notion of behavior and equivalence for them, while execution traces are perfectly adequate for deterministic systems. Debugging non-deterministic systems can be a nightmare since transient bugs may not be reproduced. Analyzing systems is also much more difficult since the state space tends to explode. Therefore, it is important to reserve nondeterminism for places where it is really mandatory, i.e., interactive systems[3], and to forget about it for reactive systems. Historically, it was long thought that concurrency and non-determinism had to go together. [...] The main merit of synchronous languages is probably to have reconciled concurrency and determinism.

The DSLs for this class of systems that Berry (1989) called for are thoroughly surveyed by Benveniste et al. (2003). Here we content ourselves with but a taste.

[3] An *interactive system* is one that takes control of the interaction. Berry cites operating systems, databases and the internet as examples.

A central strand in this tradition is concerned with *synchronous dataflow*, or what might loosely be thought of as generalised circuits. The canonical such language is Lustre (Halbwachs, Caspi, Raymond, and Pilaud 1991) which extends the simple semantics of §4.1 with a notion of sampling: values can be present or absent at each instant. (In a constructive circuit all values are always present.) *Clocks* are used to statically guarantee that a signal is used only when it is present, which ensures that the corresponding Kahn network can be implemented with finite buffers (Caspi 1992). Note that these do not coincide with a hardware designer's notion of clock as they need not be periodic. A variant of Lustre that included some constructs for expressing floorplans was proposed for hardware design (Rocheteau and Halbwachs 1991).

More recently there has been an effort to lift the features of ML to this synchronous dataflow paradigm. Higher-order functions have been treated by Caspi and Pouzet (1998) and Colaço, Girault, Hamon, and Pouzet (2004), and pattern matching by Hamon (2006), resulting in the language Lucid Synchrone. Here clocks are formalised as types. The language also supports hierarchical state machines. The compiler can optionally ensure that a program has a finite-state implementation using a simple test that is sound but not complete. Caspi and Pouzet observe that this work connects synchrony to the deforestation techniques of Wadler (1990) for functional programs.

The other main thread of the synchronous language tradition is the imperative paradigm as exemplified by Esterel (Potop-Butucaru et al. 2007). Sequential and parallel composition are provided, and the usual battery of control constructs including loops and exception handling as well as some specialised ones such as preemption and suspension. Communication is provided by signals which are broadcast within a scope; in each instant they are either present or absent. A semantics of Esterel is given by translation into the constructive circuits that we discussed in §4.1, whose theory was developed for just this purpose.

The synchronous languages share many issues with hardware design. For instance, finite-state machines that are reactive (responding at every instant, also termed *input enabled* by process algebraists) or deterministic individually may in combination lose these properties (Maraninchi and Halbwachs 1996). This issue is subsumed by the notion of *causality*, that of determining when a variable contains a valid value and what that value is. In the traditional circuit semantics of §4.1, causality is ensured by the dictum that "all loops must contain a delay". (Similarly the notion of *guardedness* in process algebra is a causal notion (Milner 1989).) The clocks of the synchronous dataflow languages ensure this kind of safety while Esterel uses a specific analysis.

In contrast to behavioural synthesis, these languages are more predictable: timing behaviour is manifest in the source text, and all constructs are deterministic. As for circuits, the assumption of synchrony allows worst-case timing analysis to be performed separately from the logical design.

4.3.2 Algebraic Techniques

We briefly survey some algebraic approaches to describing circuits: the first two are in the tradition of process algebra, and the last algebraic specification. Where the functional programming techniques discussed earlier emphasise higher-level structure, these languages can be seen as providing alternative notation and semantics for the circuits themselves.

Cardelli and Plotkin 1982; 1981 adapted (what became) Milner's SCCS 1983; 1989 into a "high level chip assembly language" – a notation for describing circuits and layouts purely structurally. This language is deeply embedded into ML, which serves as a metalanguage for composition and parametrisation. A continuous-time behavioural semantics for circuits is given at a much lower level than our synchronous one. Recently Park and Im (2011) have developed a linearly-typed higher-order functional notation for a similar purpose.

Milne (1985) developed the process algebra CIRCAL in the same tradition. Both synchronous and asynchronous systems can be treated through the judicious introduction of non-deterministic choice. Due to its semantic neutrality it can be used at all levels of abstraction, which can be connected by refinement relations. More recently it has been extended to reconfigurable hardware (Milne 2006).

The FUNNEL compiler of Stavridou (1993) translates circuits expressed as recursion equations into the algebraic specification language OBJ, with the goal of specifying, simulating and verifying them. One could consider OBJ to be a functional programming language where higher-order functions have been sacrificed for very powerful reasoning principles, such as equational rewriting and fully-automatic proofs by induction. The ACL2 theorem prover used by Hunt Jr. and his collaborators to verify various microprocessors has made a similar trade off (Hunt Jr., Swords, Davis, and Slobodova 2010).

As OBJ itself is first-order, sequential behaviour was initially modelled as a global history, with sets of tuples of the form $(w, value, time)$ where w is some enumeration of wires, *time* is a natural number and *value* is a Boolean (Stavridou 1993, §4.3.3). Later a mild extension to OBJ allowed the use of pseudo-second order functions, yielding "a powerful first-order calculus for reasoning about first-order functions" that could represent sequential behaviour directly. We note that both approaches preclude the use of circuit combinators (§4.2.1) as these are even higher-order.

Stavridou (1994) applied these techniques to "Gordon's computer", a standard example for mechanical verification of hardware, and also reviews other equational approaches to describing circuits.

4.3.3 Relational models

A reason to shift away from functions is to avail the designer of the traditional top-down program development methodology based on refinement (de Roever and Engelhardt 1998), where a specification is transformed into a more deterministic and detailed artifact expressed in the same language. Sheeran (1990) followed this train of thought when proposing a relational calculus of circuits called Ruby. Here combinational circuits and their specifications are taken to be strongly-typed relations on instantaneous values, with streams of such values used for sequential networks. As in μFP, higher-order circuit combinators are given geometric interpretations.

The ultimate result of refinement in Ruby is a *causal* relation, which are those that are functionally determined in a way familiar from database theory and logic programming: there must exist a partitioning of the fields of all relations into *inputs* and *outputs* where the latter is determined by the former. This excludes the bidirectional dataflow of busses and MOS circuits which are naturally modelled relationally. The T-Ruby system of Sharp and Rasmussen (1997) can simulate and generate synthesisable VHDL for this subset.

Ruby has been applied to similar systems as μFP – regular and arithmetic circuits (Jones and Sheeran 1993), and innovatively, butterflies such as FFTs. However as we saw with μFP, the purely combinatory style can make for awkward descriptions. Indeed the Lava approach, with its extensive battery of testing and verification tools and ad hoc combining forms, has shown that supporting exploration with instant feedback trumps formal dexterity during the design process. The recent Wired project (Axelsson et al. 2005) combines these themes in a language for capturing very low-level properties of chip design.

We note the extensive literature on modelling circuits in a higher-order logic (Camilleri et al. 1986) (etc.) but it takes us too far afield to review it here.

4.3.4 Other models of "boxes and wires"

Another mode of generalisation is to focus on general ways of composing "boxes and wires" diagrams, and investigate their equational properties. Category theorists claim that these find their natural expression as some kind of *monoidal category*, and indeed these structures and their "string diagrams" have been surveyed at length by Selinger (2011).

These models are constructed using combinators, and therefore suffer from the plumbing problem. Braibant (2011) models circuits in the Coq proof assistant using such an approach and it is clear that while the algebra is pleasant one would struggle to comprehend the syntactic expression of a circuit without an accompanying diagram. This tension has been substantially resolved for a particular set of combinators – the Arrows of Hughes (2000) – by the notation of Paterson (2001), which allows us to write pointwise or point-free definitions at our discretion. These form the basis of our approach, which we discuss at length in the rest of this thesis.

The Hume project has developed a "box calculus" (Grov and Michaelson 2010) that supports the refinement of computational boxes connected by wiring described in a finite-state coordination language.

4.3.5 On formal functional models for synchronous digital circuits

To reason about our circuits using a proof assistant, we need an accurate formal model for them. Here we discuss a few of the traditional models.

In general we wish to reason in two ways. Firstly we would like to transform our circuits using equational reasoning, and as we saw above the domain models support this mode very well; such techniques scale easily as they are largely independent of the size of the state space. Secondly we wish to show that particular circuits have specific properties, for which temporal logic in general (Manna and Pnueli 1992), and its automation in the form of model checking (Clarke et al. 1999), has proven very successful. However as observed by Matthews (2000, §7.6), by encapsulating state our stream models sometimes make assertions more difficult to write than in explicit-state formalisms. Day, Aagaard, and Cook (2000) discuss moving between these representations for a shallowly-embedded HDL.

Most systems we discuss here implicitly appeal to the *synchronous isomorphism*:

$$\mathsf{Signal}\ (\alpha,\ \beta)\ \simeq\ (\mathsf{Signal}\ \alpha,\ \mathsf{Signal}\ \beta)$$

where $\mathsf{Signal}\ \alpha$ is a type that captures the temporal behaviour of a wire. Intuitively this characterises systems with non-blocking components that communicate in globally-synchronised rounds; it requires functions $\mathsf{Signal}\ \alpha \rightarrow \mathsf{Signal}\ \beta$ to be length preserving, which clearly does not hold in asynchronous settings.

This isomorphism underpins laws that allow stateful components to be combined and decomposed, such as the one shown in §4.2.1. As we observed there, our $\mathsf{Signal}\ \alpha$ domain of Figure 4.1 does not not satisfy this isomorphism as it contains *junk* in the form of partial streams x_0 :> ... :> x_n :> $\perp$, where $\perp$ is the least-defined sequence (Winskel 1993, §8.2). These preclude the definition of an injective zip. We note that Kahn networks and other domains based on prefix orders have the same deficiency.

While preferring this model, Caspi (1992) observes we could also take $\mathsf{Signal}\ \alpha$ to be some set of functions $\mathsf{nat} \rightarrow \alpha$, which supports the operations of Figure 4.1 while satisfying the synchronous isomorphism. (This is an *environment* or *reader* monad.) Unfortunately it also admits junk in the form of the non-causal functions $\mathsf{Signal}\ \alpha \rightarrow \mathsf{Signal}\ \beta$ whose behaviour at time n depends on the value of their arguments at time $m > n$. Abbott, Altenkirch, and Ghani (2005) have studied these *containers* in categorical and type-theoretic settings; see also Bertot and Komendantskaya (2008).

This attempt to identify $\mathsf{Signal}\ \alpha$ with the set of causal infinite streams over α suggests the use of *corecursion* (Coquand 1993). Such an approach was advocated by Paulin-Mohring (1995) who used it to model a multiplier and its properties in the Coq proof assistant. Caspi and Pouzet (1998) show how to compose corecursive descriptions from systems of recursion equations for their higher-order synchronous dataflow language Lucid Synchrone (see §4.3.1), but it is unclear that it can be used in proof assistants where corecursive definitions are typically required to take particular syntactic forms. Such constraints guarantee *productivity* of the definition and hence well-definedness of the sequence. Note these also rule out the use of higher-order combinators such as those in §4.2.

The literature on models of dataflow and streaming computation is too vast to review here; we only point to some closely related recent work. Hughes, Pareto, and Sabry (1996), Barthe, Frade, Giménez, Pinto, and Uustalu (2004) and Abel (2010) propose *sized types* as a compositional way of ensuring productivity. The "fast and loose reasoning" of Danielsson, Hughes, Jansson, and Gibbons (2006) does not apply to unstructured recursion equations, though some may consider a unique fixed-point property (Hinze and James 2011) to be something of a replacement; see also the work of Matthews (1999) mentioned in §4.2.7. Broy and Stølen (2001) use prefix-ordered domains to specify *interactive* systems. Möller and Tucker (1998) provide further pointers to formal stream-based models for hardware.

4.4 Concluding remarks

Here we have focused on surveying how functional programming has been used to describe, design and validate synchronous hardware. Jantsch and Sander (2005) situate this *model of computation* in a spectrum of those relevant to the construction of embedded systems, including the codesign of hardware and software. The reader can find surveys of HDLs in other styles in McEvoy and Tucker (1990a), Stavridou (1993, Chapter 3) and Claessen (2001, §1.11), while Johnson (1983, Chapter 1) and Sheeran (2005) provide more historical perspective on the early days of this tradition. Sharing in EDSLs is discussed at length by Kiselyov (2011).

The central goal of all of these systems is to make higher-assurance hardware easier to design, and to find a good trade-off between formal rigour and ease of use. This is a problem of increasing interest as FGPAs and other reprogrammable logic becomes commonplace (Cardoso, Diniz, and Weinhardt 2010), and it is not always feasible to fully verify custom hardware structures for computation kernels, or coprocessors like the Reduceron (§4.2.6). Hope may lie in automatic state-space traversal techniques (Clarke et al. 1999), but these too require expertise quite distant from hardware design. Random testing as epitomised by QuickCheck (Claessen 2001, Chapter 4) is an alternative that works well when effects can be tamed, as they are in a purely functional setting.

In contrast proof assistants are essential to the verification of complex designs and the refinement processes advocated by Johnson (2001), and indeed Intel's Integrated Design and Validation (IDV) system appears to have successfully applied this methodology to their designs (Grundy, Melham, and O'Leary 2006; Seger, Jones, O'Leary, Melham, Aagaard, Barrett, and Syme 2005), though perhaps not as ambitiously as Johnson aspired to. Functional programming techniques underpin all large-scale verification efforts such as the ARM processor models of Fox, Gordon, and Myreen (2010) and the x86-compatible models of Hunt Jr. et al. (2010).

The systems presented above are all experimental, both in their methodology and the artifacts described with them. Sheeran (2011) has used her various platforms to explore different kinds of circuits, and shown that rapid feedback in the form of simulation, testing and model checking is most valuable to the designer. Johnson and Bose (1997) and Seger et al. (2005) make similar observations about their refinement efforts. This is clear evidence that functional programming techniques are a useful substrate for this diverse range of tasks.

The algebraic structure of circuits has much in common with other forms of parallel and distributed programming, which also use parallel prefix (or scan) networks (Sheeran 2011), and butterflies and other networks that are naturally rendered using powerlists (Paterson 2003). These structures link our domain with the search for higher-level programming abstractions for historically arcane DSP and GPU architectures (Axelsson, Claessen, Sheeran, Svenningsson, Engdal, and Persson 2010; Chakravarty, Keller, Lee, McDonell, and Grover 2011b; Sweeney 2009) and multicore systems (Keller, Chakravarty, Leshchinskiy, Peyton Jones, and Lippmeier 2010). Singh (2007) also proposes adopting concurrency abstractions developed in functional programming settings to hardware.

As we discussed in §4.2.2 and §4.2.10, functional programming has been used as a basis for behavioural synthesis. Recently Harrison, Procter, Agron, Kimmell, and Allwein (2009) propose to extend Johnson's use of Wand's compiler/virtual machine split (§4.2.2) to a concurrent language by using a resumption monad; every element of this agenda poses difficulties for other programming techniques due to their lack of types, higher-order facilities or controlled effects.

Another quintessential dimension of this tradition is the development of increasingly fancy type systems (Chakravarty et al. 2005; Diatchki, Jones, and Leslie 2005; Kaes 1988; Peyton Jones, Vytiniotis, Weirich, and Shields 2007; Wadler and Blott 1989) (etc.) that are comfortable to program with. Such techniques have already been shown useful for parametrising circuit generators by vector widths (§4.2.6). Sheard (2007) proposes his Ωmega language as a vehicle for exploring the use of this machinery in great generality; one eventually might hope to write circuit generators as resource-aware *active libraries* (Sheeran 2011; Veldhuizen 2004).

Sheard also argues that HDLs should formally recognise the distinction between circuits and their generators; in other words, the *staging* of descriptions should be manifest, which is certainly necessary to resolve the semantic tensions we saw throughout §4.2. Kiselyov, Swadi, and Taha (2004) and Gillenwater, Malecha, Salama, Zhu, Taha, Grundy, and O'Leary (2010) demonstrate how this idea works in practice. Megacz (2011) is pursuing an approach in which two-level programs with first-order object expressions are flattened into single-level programs which represent object language terms using a generalization of the Arrow programming abstraction due to Hughes (2000).

We also find an argument for meta-programming from the formal reasoning community, where Grundy et al. (2006) have developed two functional languages for representing circuits in a higher-order logic. These involve reification of descriptions into the logic, and not just execution; while this leads to semantic difficulties in a programming setting (Taha 2000), it is quite desirable in a proof assistant.

The limited domain of circuits and fixed-network stream processors often admits appealing diagrammatic representations which can be much easier to reason about than the expressions they visualise, as we saw in §4.2.1. This is not too surprising as effective circuits need to be mapped to floorplans. What is surprising is that while semantically-wellfounded graphical tools for first-order languages abound (André and Peraldi-Frati 2000; Harel 2009; Maraninchi and Rémond 2001) (etc.), there is a lack of support for the kind of higher-order programming advocated here.

In closing we observe the renewed interest in functional programming techniques for software due to the increasing use of parallelism and concurrency, and expect to see a similar resurgence in the context of hardware design.

Chapter 5

Arrows for synchronous digital circuits

We describe our knowledge-based programs and their scenarios as digital synchronous circuits. We structure our implementation using *Arrows*[1], a functional programming abstraction due to Hughes (2000), which allows us to capture the information flowing between the components of the system in a pure way. This is their main advantage over the Monadic approaches discussed in the previous chapter.

We outline our circuit Arrows in this chapter and demonstrate their particular strengths when we implement the knowledge-based programming constructs in the next.

5.1 What are Arrows?

Intuitively Arrows are generalised functions that support the kinds of effects that made Monads famous. This section is a modest overview that develops this intuition, assuming that the reader is familiar with modern Haskell programming techniques; those in need of further background are encouraged to consult Hughes (2000, 2004) and Paterson (2003). We will make extensive use of type classes (Kaes 1988; Wadler and Blott 1989).

The central goal of Hughes (2000) was to develop a technique for composing effectful computations that allowed their static structure to be analysed. He noted that the *bind* operator for a Monad is asymmetric:

$$(\gg\!=) :: \mathsf{Monad}\ m \Rightarrow m\,\alpha \to (\alpha \to m\,\beta) \to m\,\beta$$

which means that bind cannot examine its second argument before applying it to the result of running the first computation. The obvious attempt at a symmetric Monadic operation:

$$(\gg) :: \mathsf{Monad}\ m \Rightarrow m\,\alpha \to m\,\beta \to m\,\beta$$

prevents the second computation from depending on the value yielded by the first. Therefore a putative Arrow structure with such a composition operator must internalise enough of the

[1] Here we capitalise “Arrows” and “Monads” to emphasise that these are specific programming abstractions.

ambient λ-calculus that Monads use to pass values around but not so much that it cannot support the desired kind of static analysis. Such a move is familiar from the various combinator encodings of the λ-calculus; for example, to give a categorical semantics (Gunter 1992, Chapter 3), and for compilation (Peyton Jones 1987, Chapter 16); (Curien 1994). More broadly *point-free style* (Backus 1978) encourages this reliance on explicit plumbing by emphasising function composition over application.

Hughes defines an Arrow by a type constructor taking two arguments, schematically denoted $\alpha \rightsquigarrow \beta$, to be understood as a generalised function from α to β which possibly engages in some effects. The canonical exemplar Arrows are the pure functions of type $\alpha \to \beta$ and the *Kleisli Arrows* with type $\alpha \to m\,\beta$ that arise from arbitrary Monads m. The type constructor $(\rightsquigarrow)$ needs to support the following operations.

Firstly, in the same way that $\mathsf{return} :: \mathsf{Monad}\ m \Rightarrow \alpha \to m\,\alpha$ naturally lifts a value of type α into a Monadic value, or *computation*, the function:

$$\mathsf{arr} :: \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow (\alpha \to \beta) \to (\alpha \rightsquigarrow \beta)$$

injects Haskell's pure function space into the Arrow. (As is the situation with Monads, the converse inclusion is a per-Arrow concern.) These pure Arrows perform the critical task of plumbing values around, as we will see.

The symmetric "sequential composition" operator corresponding to $(\gg\!=)$ has type:

$$(\ggg) :: \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow (\alpha \rightsquigarrow \beta) \to (\beta \rightsquigarrow \gamma) \to (\alpha \rightsquigarrow \gamma)$$

While this operation clearly resolves the problem we had in composing Monadic computations with $(\gg)$, alone it is insufficient for composing Arrows in general.

Consider, for example, computing the average of a list of integers:

$$\begin{array}{l}\mathsf{average} :: [\mathsf{Int}] \to \mathsf{Int} \\ \mathsf{average}\ xs = \mathsf{sum}\ xs\ \text{`}\mathsf{div}\text{`}\ \mathsf{length}\ xs\end{array}$$

This involves a non-linear use of xs. Imagining that the three operations used here are instead implemented as the Arrows

$$\begin{array}{l}\mathsf{divA} :: \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow (\mathsf{Int}, \mathsf{Int}) \rightsquigarrow \mathsf{Int} \\ \mathsf{lengthA} :: \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow [\alpha] \rightsquigarrow \mathsf{Int} \\ \mathsf{sumA} :: \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow [\mathsf{Int}] \rightsquigarrow \mathsf{Int}\end{array}$$

the corresponding Arrow $\mathsf{averageA}$ cannot be implemented with only arr and $(\ggg)$; we need a "parallel composition" operation $(\ast\!\ast\!\ast)$ to shuffle values past computations:

$$(\ast\!\ast\!\ast) :: \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow (\alpha \rightsquigarrow \beta) \to (\alpha' \rightsquigarrow \beta') \to ((\alpha, \alpha') \rightsquigarrow (\beta, \beta'))$$

We use the pure Arrow $\mathsf{dupA} = \mathsf{arr}\ (\lambda x.\ (x,\ x))$ to suitably duplicate xs:

$$\begin{aligned}&\mathsf{averageA} :: \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow [\mathsf{Int}] \rightsquigarrow \mathsf{Int}\\&\mathsf{averageA} = \mathsf{dupA} \ggg (\mathsf{sumA} *\!*\!* \mathsf{lengthA}) \ggg \mathsf{divA}\end{aligned}$$

We can recover average by appealing to the $(\rightarrow)$ Arrow instance. We note that the Monadic operation that performs the same service as $(*\!*\!*)$ is always definable for Monads expressed in Haskell.

Note that we switch to *uncurried* functions: we have to bundle all the arguments into tuples, rather than returning functions as is customary in Haskell. This is because proper Arrows (those that are not Kleisli Arrows) are not Cartesian closed, that is, they do not admit "Arrow application" analogous to function application:

$$\begin{aligned}&\textbf{class}\ \mathsf{ArrowApply}\ (\rightsquigarrow)\ \textbf{where}\\&\quad \mathsf{app} :: (\alpha \rightsquigarrow \beta,\ \alpha) \rightsquigarrow \beta\end{aligned}$$

Hughes showed that if an Arrow supports ArrowApply then it is a Monad, and for these the Arrow framework provides no benefit. This lack of closure is the essence of capturing information flow using Arrows, and hence underpins our knowledge-based circuits infrastructure; see §6.1.

As Haskell is intended to have a deterministic sequential semantics, Hughes (2000) suggests that $(*\!*\!*)$ be derived from the asymmetric combinator first:

$$\mathsf{first} :: \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow (\alpha \rightsquigarrow \beta) \rightarrow ((\alpha,\ \gamma) \rightsquigarrow (\beta,\ \gamma))$$

so that the order of effects is inherited from $(\ggg)$. Thus the Arrow class is defined[2]:

$$\begin{aligned}&\textbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow)\ \textbf{where}\\&\quad \mathsf{arr} :: (\alpha \rightarrow \beta) \rightarrow (\alpha \rightsquigarrow \beta)\\&\quad (\ggg) :: (\alpha \rightsquigarrow \beta) \rightarrow (\beta \rightsquigarrow \gamma) \rightarrow (\alpha \rightsquigarrow \gamma)\\&\quad \mathsf{first} :: (\alpha \rightsquigarrow \beta) \rightarrow ((\alpha,\ \gamma) \rightsquigarrow (\beta,\ \gamma))\end{aligned}$$

We will also make use of the *fanout* combinator:

$$\begin{aligned}&(\&\&\&) :: \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow (\alpha \rightsquigarrow \beta) \rightarrow (\alpha \rightsquigarrow \gamma) \rightarrow (\alpha \rightsquigarrow (\beta,\ \gamma))\\&f\ \&\&\&\ g = \mathsf{dupA} \ggg f *\!*\!* g\end{aligned}$$

which feeds a single input to f and g and pairs their output, and abbreviate $\mathsf{arr\ id}$ by $\mathsf{returnA}$.

As anyone who has attempted to understand Haskell one-liners knows, writing point-free programs is one thing, understanding them is something else, and maintenance is never mentioned. Fortunately Paterson (2001) has developed a notation that substantially improves the readability of Arrow code, largely by computing the plumbing of values between computations for us. For

[2] This class has since been split into the Category and Arrow classes. We ignore this complication.

$$
\begin{array}{l}
\mathsf{mapA} :: \mathsf{ArrowChoice}\ (\rightsquigarrow) \\
\qquad \Rightarrow (\alpha \rightsquigarrow \beta) \\
\qquad \to ([\alpha] \rightsquigarrow [\beta]) \\
\mathsf{mapA}\ f = \mathbf{proc}\ xxs \to \\
\quad \mathbf{case}\ xxs\ \mathbf{of} \\
\qquad [\,] \to \mathsf{returnA} \prec [\,] \\
\qquad x : xs \to \\
\qquad\ \mathbf{do}\ y \leftarrow f \prec x \\
\qquad\quad\ ys \leftarrow \mathsf{mapA}\ f \prec xs \\
\qquad\quad\ \mathsf{returnA} \prec y : ys
\end{array}
\qquad
\begin{array}{l}
\mathsf{mapAC} :: \mathsf{Arrow}\ (\rightsquigarrow) \\
\qquad \Rightarrow ((\gamma, \alpha) \rightsquigarrow \beta) \\
\qquad \to ((\gamma, [\alpha]) \rightsquigarrow [\beta]) \\
\mathsf{mapAC}\ f = \mathbf{proc}\ (env, xxs) \to \\
\quad \mathbf{case}\ xxs\ \mathbf{of} \\
\qquad [\,] \to \mathsf{returnA} \prec [\,] \\
\qquad x : xs \to \\
\qquad\ \mathbf{do}\ y \leftarrow f \prec (env, x) \\
\qquad\quad\ ys \leftarrow \mathsf{mapAC}\ f \prec (env, xs) \\
\qquad\quad\ \mathsf{returnA} \prec y : ys
\end{array}
$$

Figure 5.1: The mapA and mapAC combinators.

example, averageA can be written as the program on the left, from which Paterson's `arrowp` pre-processor generates the code on the right:

$$
\begin{array}{l}
\mathsf{averageA} :: [\mathsf{Int}] \to \mathsf{Int} \\
\mathsf{averageA} = \mathbf{proc}\ xs \to \\
\quad \mathbf{do}\ s \leftarrow \mathsf{sum} \prec xs \\
\qquad l \leftarrow \mathsf{length} \prec xs \\
\qquad \mathsf{returnA} \prec s \text{ `div` } l
\end{array}
\qquad
\begin{array}{l}
\mathsf{averageA} = (\mathsf{arr}\ (\lambda xs.\ (xs, xs)) \ggg \\
\qquad\qquad (\mathsf{first}\ \mathsf{sum} \ggg \mathsf{arr}\ (\lambda(s, xs).\ (xs, s))) \ggg \\
\qquad\qquad\ (\mathsf{first}\ \mathsf{length} \ggg \mathsf{arr}\ (\lambda(l, s).\ (s \text{ `div` } l))))
\end{array}
$$

In essence, we can write our generalised functions in either a point-free or pointed style (naming various data values), and the pre-processor converts them into well-structured point-free code. The key subtlety that prevents Arrows having a straightforward notation like Monads involves the scope of variables: λ-bound variables can be used anywhere in their scope, as usual, but Arrow-bound variables (bound with either **proc** or $\leftarrow$) can only be used to the right of a tail ($\prec$). Intuitively if we use an Arrow-bound variable in the shaft of an Arrow then we are conflating the staging distinction, requiring that the Arrow support some kind of application and hence that it be a Monad. See Paterson (2001) for the full story.

Instances of the Arrow classes are intended to satisfy several laws, which are presented in algebraic and diagrammatic forms by Paterson (2003). Using these we can lift many common recursion patterns like map to the Arrow setting, which is complicated by the lack of Cartesian closure. The following section explains the notation for higher-order Arrow programming.

5.1.1 Command combinators

Arrows support similar kinds of generic scaffolding to Monads. For instance, the mapA function shown in Figure 5.1 is an analogue of the classic map and mapM combinators. The ArrowChoice class supports a dynamic choice between two Arrows as we dicuss further in §5.2.2; see also Hughes (2004).

However as proper Arrows are not Cartesian closed, the argument f to mapA does not have access to any Arrow-bound variables apart from the elements of the list. (Recall that λ-bound variables have their usual scoping rules.) Consider, for instance, adding a given number to each element of a list:

$$\mathsf{add} :: (\mathsf{Integer}, [\mathsf{Integer}]) \to [\mathsf{Integer}]$$
$$\mathsf{add}\ (x, ys) = \mathsf{map}\ (\lambda y.\ x + y)\ ys$$

We can easily convert this function into its Monadic equivalent using the standard mapM combinator:

$$\mathsf{addM} :: \mathsf{Monad}\ m \Rightarrow (\mathsf{Integer}, [\mathsf{Integer}]) \to m\ [\mathsf{Integer}]$$
$$\mathsf{addM}\ (x, ys) = \mathsf{mapM}\ (\lambda y.\ \mathsf{return}\ (x + y))\ ys$$

This is not so readily achieved with an Arrow, and so the specialised notation supports the notion of a *command combinator*, which intuitively passes an arbitrary environment through our scaffolding combinators. By convention the first parameter is reserved for this purpose – extra arguments are paired to the right. The generalisation for mapA is shown in Figure 5.1, and using it we can define addA:

$$\mathsf{addA} :: \mathsf{Arrow}\ (\leadsto) \Rightarrow (\mathsf{Integer}, [\mathsf{Integer}]) \leadsto [\mathsf{Integer}]$$
$$\mathsf{addA} = \mathbf{proc}\ (x, ys) \to (\!|\ \mathsf{mapAC}\ (\lambda y \to \mathsf{returnA} \prec x + y)\ |\!)\ ys$$

The "banana brackets" provide a syntactic cue to the Arrow pre-processor that we wish to apply a command combinator (here mapAC) to one or more commands, and possibly arguments (here *ys*). Compare this to the case where x is λ-bound:

$$\mathsf{addA}' :: \mathsf{Arrow}\ (\leadsto) \Rightarrow \mathsf{Integer} \to ([\mathsf{Integer}] \leadsto [\mathsf{Integer}])$$
$$\mathsf{addA}'\ x = \mathsf{mapA}\ (\mathsf{arr}\ (\lambda y.\ x + y))$$

Command combinators can be used to give an expression-like syntax that is sometimes easier to read than the alternatives, and we can generically turn a standard Arrow into a command combinator with operators such as:

$$\mathsf{liftAC2} :: \mathsf{Arrow}\ (\leadsto) \Rightarrow ((\alpha, \beta) \leadsto \gamma) \to (\gamma \leadsto \alpha) \to (\gamma \leadsto \beta) \to (\gamma \leadsto \gamma)$$
$$\mathsf{liftAC2}\ op\ f\ g = f\ \&\&\&\ g \ggg op$$

$$\mathsf{liftA2} :: \mathsf{Arrow}\ (\leadsto) \Rightarrow (b \to c \to d) \to (\gamma \leadsto b) \to (\gamma \leadsto c) \to (\gamma \leadsto d)$$
$$\mathsf{liftA2} = \mathsf{liftAC2} \circ \mathsf{arr} \circ \mathsf{uncurry}$$

We append a C to the names of Arrows that are (specifically) command combinators, and will silently lift Arrows to their command combinator variants when we need to.

Hughes (2004) presents several examples in this style. The syntactic subtleties are explained in the Glasgow Haskell Compiler User Manual (§7.13).

Like Monads, Arrows are only useful for the extra operations they support beyond the structural plumbing described above that is (abstractly) common to all instances. The following section sketches the way we specify these operations, and the specifics of our circuit Arrows follow.

5.1.2 A pattern of Arrows for reinterpretation

Our goal is to adapt the generic Arrow framework to our circuits domain in such a way that we can give our descriptions multiple interpretations (§4.2.1). We aim to model our circuits as Arrows simply by replacing the function arrow (→) in the simple circuit EDSL shown in Figure 4.1 on page 75 with suitable Arrows, uncurrying as necessary.

One might hope to adapt Hughes's original motivation for Arrows to the goal of reinterpretation, obtaining a netlist "statically" and a simulation semantics "dynamically". In particular Hughes gave an abstract interface to the parser combinators of Swierstra and Duponcheel (1996) using the following Parser Arrow:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{StaticParser}\ s = \mathsf{SP}\ \mathsf{Bool}\ [s] \\
&\textbf{newtype}\ \mathsf{DynamicParser}\ s\ \alpha\ \beta = \mathsf{DP}\ ((\alpha,\ [s]) \to (\beta,\ [s])) \\
&\textbf{data}\ \mathsf{Parser}\ s\ \alpha\ \beta = \mathsf{P}\ (\mathsf{StaticParser}\ s)\ (\mathsf{DynamicParser}\ s\ \alpha\ \beta)
\end{aligned}
$$

where s is the type of symbols. The idea is that $\mathsf{StaticParser} s$ describes the preconditions for invoking $\mathsf{DynamicParser}\ s\ \alpha\ \beta$ – that is, whether it accepts the empty string, and which tokens it accepts first. The generic Arrow plumbing described above can safely propagate this information in a way that Monadic operations cannot.

Unfortunately we cannot obtain a netlist using this approach, as the structure of an Arrow network is only partially "static"; we cannot analyse the Arrow plumbing, which is polymorphic and allows for arbitrary transformations on values using the arr combinator. For example, as we cannot even determine what a function f of type $(\alpha,\ \alpha) \to (\alpha,\ \alpha)$ does from within the language we have no way of knowing what the netlist for $\mathsf{arr}\ f$ should be.

With these observations in mind we evolve a typical Monadic type-class-based abstraction, following Bjesse et al. (1998); Erkök (2002); Matthews et al. (1998); Paterson (2003) amongst others, to an Arrow setting. This is our starting point:

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Bit} = \mathsf{Bool} \\
&\textbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{Circuit}\ m\ sig\ \textbf{where} \\
&\quad \mathsf{and2} :: (sig\ \mathsf{Bit},\ sig\ \mathsf{Bit}) \to m\ (sig\ \mathsf{Bit}) \\
&\quad \mathsf{neg} :: sig\ \mathsf{Bit} \to m\ (sig\ \mathsf{Bit}) \\
&\quad \mathsf{delay} :: \alpha \to sig\ \alpha \to m\ (sig\ \alpha)
\end{aligned}
$$

Intuitively the class contains an adequate set of gates, with the Monad m providing any effects we need to interpret the circuit, and the sig type constructor providing the temporal structure. Specifically we can recover our simple simulation semantics of Figure 4.1 by taking m to be the identity Monad with type constructor $\mathsf{IdM}\ \alpha = \alpha$ and sig to be our signal type, i.e. $sig\ \alpha = \mathsf{Signal}\ \alpha$. A netlist could use a state Monad:

$$
\textbf{newtype}\ \mathsf{StateM}\ s\ \alpha = \mathsf{StateM}\ (s \to (\alpha,\ s))
$$

with the state providing a name supply for identifying the components, and *sig* α defined to be such a name; a value "flowing on a wire" is the identity of the component that drives it.

Our first step is to replace the Monad with an Arrow:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow)\ \Rightarrow\ \mathsf{Circuit}\ (\rightsquigarrow)\ \textbf{where} \\
\quad \mathsf{andA} :: (\mathsf{Bit}, \mathsf{Bit}) \rightsquigarrow \mathsf{Bit} \\
\quad \mathsf{notA} :: \mathsf{Bit} \rightsquigarrow \mathsf{Bit} \\
\quad \mathsf{delayA} :: \alpha \to (\alpha \rightsquigarrow \alpha)
\end{array}
$$

We simultaneously eliminate the *sig* parameter by incorporating temporal behaviour into the Arrow, exploiting the "morally correct" synchronous isomorphism (§4.3.5):

$$(sig\ \alpha,\ sig\ \beta)\ \simeq\ sig\ (\alpha,\ \beta)$$

Once again, a simulation Arrow might take $\alpha \rightsquigarrow \beta$ to be $[\alpha] \to [\beta]$, and the netlist can use the Kleisli Arrow $\alpha \to \mathsf{StateM}\ s\ \beta$. Consequently we need to directly reinterpret Bit, which motivates the following multi-parameter type class (MPTC):

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow)\ \Rightarrow\ \mathsf{Circuit}\ (\rightsquigarrow)\ bit\ \textbf{where} \\
\quad \mathsf{andA} :: (bit, bit) \rightsquigarrow bit \\
\quad \mathsf{notA} :: bit \rightsquigarrow bit \\
\quad \mathsf{delayA} :: \alpha \to (\alpha \rightsquigarrow \alpha)
\end{array}
$$

Unfortunately all uses of delayA are ambiguous as it does not constrain the *bit* parameter. A partial (and in our case, adequate) solution is to require that each of the members of the class must mention all of the type variables in the head of the class declaration. We note that this does not completely resolve this issue as there can still be read ∘ show-type ambiguities where a constrained type variable is not present in the type of the expression. See Odersky, Wadler, and Wehr (1995) for further discussion on this point.

Implicit in the first two definitions were the constructors of the Bit type, which we need to make manifest for the abstract type *bit*. Therefore we arrive at these definitions:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow)\ \Rightarrow\ \mathsf{Circuit}\ (\rightsquigarrow)\ bit\ \textbf{where} \\
\quad \mathsf{falseA} :: ()\ \rightsquigarrow bit \\
\quad \mathsf{trueA} :: ()\ \rightsquigarrow bit \\
\\
\quad \mathsf{andA} :: (bit, bit) \rightsquigarrow bit \\
\quad \mathsf{notA} :: bit \rightsquigarrow bit \\
\textbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow)\ \Rightarrow\ \mathsf{Delay}\ (\rightsquigarrow)\ \textbf{where} \\
\quad \mathsf{delayA} :: \alpha \to (\alpha \rightsquigarrow \alpha)
\end{array}
$$

The key invariant we require of all instances of these classes is that there are no user-visible operations on the types used to instantiate *bit* that distinguish its values. For instance an interpretation using *bit* = Bool allows the pure Arrow arr (uncurry (&&)) to be silently mixed with uses of andA, defeating our attempts at reinterpretation. Similarly the Arrow cannot be the bare function type constructor (→).

Provided interpretations and circuits respect this invariant pure Arrows can only have innocuous behaviour. We discuss this further in §5.6.

We note that these particular classes are not ideal to program with; for instance, delayA cannot be initialised at the *bit* type using falseA and trueA! We discuss pragmatics in the following sections.

This style of reinterpretation has been named "finally tagless" by Carette et al. (2009); we mildly extend their work by allowing representations of types such as Bit to vary with the interpretation. It can be seen as a partial solution to the expression problem as popularised by Wadler: this use of type classes gives us an "open" syntax in the sense that we can define more constructs simply by defining more classes, and interpret these constructs without disturbing the existing instances. Some extensions (such as non-determinism and circuit probes) may involve significant renovation of the Arrow's representation, however.

5.2 Circuit Arrows

We now sketch our circuit-specific Arrows using the technique of the previous section to specify them. This framework forces us to clearly distinguish amongst:

- values that flow along the wires;
- operations connected by these wires; and
- circuit generators that create networks of operations connected by wires.

As observed by Erkök (2002, p9), a key intuition is to determine when effects are used: do they occur at "run time" – during the interpretation of the circuit – or while generating it? The former should be circuit Arrows, while the latter can be deferred to the general mechanisms of Haskell.

In our case, the basic effect is a unit-time delay operation, from which we can build the finite memories of actual circuits. We are also interested in other effects such as probes and non-determinism (§5.2.5), and later, knowledge (§6.1).

5.2.1 The ArrowComb class

We begin with the foundational ArrowComb class:

class Arrow (⇝) ⇒ ArrowComb (⇝) **where**

$$
\begin{aligned}
&\textbf{type}\ \mathsf{B}(\rightsquigarrow)\ ::\ *\\
&\mathsf{falseA}\ ::\ \gamma \rightsquigarrow \mathsf{B}(\rightsquigarrow)\\
&\mathsf{trueA}\ ::\ \gamma \rightsquigarrow \mathsf{B}(\rightsquigarrow)\\
&\mathsf{andA}\ ::\ (\mathsf{B}(\rightsquigarrow), \mathsf{B}(\rightsquigarrow)) \rightsquigarrow \mathsf{B}(\rightsquigarrow)\\
&\mathsf{notA}\ ::\ \mathsf{B}(\rightsquigarrow) \rightsquigarrow \mathsf{B}(\rightsquigarrow)\\
&\mathsf{note}\ ::\ \mathsf{String} \to (b \rightsquigarrow c) \to (b \rightsquigarrow c)\\
&\mathsf{note}\ \_\ =\ id
\end{aligned}
$$

This is essentially the final definition of §5.1.2 with the trivial addition of a note function that is useful for delineating subcircuits (such as in the netlist interpretation of §5.4.1). We use the *associated type* (Chakravarty et al. 2005) $\mathsf{B}(\rightsquigarrow)$ instead of the type-class parameter *bit*, which has the effect of making the type of bits a function of the Arrow $(\rightsquigarrow)$. This eliminates the ambiguity that would need to be resolved by the user at the cost of only allowing one type of bit per Arrow.

In practice we add many redundant combinational gates and define standard logical syntax for their command-combinator variants.

5.2.2 The ArrowMux class

A standard combinational circuit component is the *multiplexer*, which outputs one of its two data inputs depending on a separate selection input. We might hope to press the standard ArrowChoice class into service as it would allow us to use the **case** and **if** control constructs in the Arrow notation. That class defines the (|||) operator that does to sum types what (∗∗∗) does to products:

$$
\begin{aligned}
&\textbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow)\ \Rightarrow\ \mathsf{ArrowChoice}\ (\rightsquigarrow)\ \textbf{where}\\
&\quad \ldots\\
&\quad (|||)\ ::\ (\alpha \rightsquigarrow \gamma) \to (\beta \rightsquigarrow \gamma) \to (\mathsf{Either}\ \alpha\ \beta \rightsquigarrow \gamma)
\end{aligned}
$$

Unfortunately this does not support reinterpretation as our interpretations may not be able to represent arbitrary polymorphic types. Moreover the semantics of synchronous circuits require us to clock (execute) both branches of the choice even if they are unselected, as they may update internal state. This would involve manufacturing a value of type α or β not provided by the input, which we cannot parametrically do. We might say that ArrowMux implements a form of clock gating.

For these reasons we need to provide our own choice combinator:

$$
\begin{aligned}
&\textbf{class}\ \mathsf{ArrowComb}\ (\rightsquigarrow)\ \Rightarrow\ \mathsf{ArrowMux}\ (\rightsquigarrow)\ \alpha\ \textbf{where}\\
&\quad \mathsf{muxA}\ ::\ (\mathsf{B}(\rightsquigarrow), (\alpha, \alpha)) \rightsquigarrow \alpha
\end{aligned}
$$

with the expectation that muxA outputs the first value if the condition is true, and the second otherwise. We include the type α in the head of the type class to support the generics of §5.3.

5.2.3 The ArrowDelay class

We need a primitive delay operation to support sequential circuits. As observed in §5.1.2 the statically-initialised operation:

$$\mathsf{delayA} :: \alpha \to (\alpha \rightsquigarrow \alpha)$$

cannot be initialised at type $\mathsf{B}(\rightsquigarrow)$ as we have no way of feeding the result of the constant Arrows to the delayA function. (Recall that we can embed pure functions in Arrows, but not necessarily the other way around.) For this reason we ask for this operation:

$$\begin{array}{l} \textbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow \mathsf{ArrowDelay}\ (\rightsquigarrow)\ \alpha\ \textbf{where} \\ \quad \mathsf{delayA} :: (\alpha, \alpha) \rightsquigarrow \alpha \end{array}$$

The first value is yielded by delayA in the first instant, and at later instants delayA yields the second value from the previous instant; the first argument is ignored at later times. The command combinator variant:

$$\begin{array}{l} \mathsf{delayAC} :: \mathsf{ArrowDelay}\ (\rightsquigarrow)\ \alpha \Rightarrow (\gamma \rightsquigarrow \alpha) \to (\gamma \rightsquigarrow \alpha) \to (\gamma \rightsquigarrow \alpha) \\ \mathsf{delayAC} = \mathsf{liftAC2}\ \mathsf{delayA} \end{array}$$

is essentially the followed-by, or initialised delay, operator -> of Lustre (see §4.3.1 and Halbwachs et al. (1991)). We again use an MPTC to support the generics of §5.3.

5.2.4 The ArrowCombLoop class

We expect our interpretations to provide instances of the ArrowLoop class defined by Paterson (2001):

$$\begin{array}{l} \textbf{class}\ \mathsf{ArrowLoop}\ (\rightsquigarrow)\ \textbf{where} \\ \quad \mathsf{loop} :: ((\alpha, \gamma) \rightsquigarrow (\beta, \gamma)) \to (\alpha \rightsquigarrow \beta) \end{array}$$

However as we discussed in §4.1, we can only expect these instances to work when the cycle includes a delay, in the classic sequential circuit tradition; this allows the use of the very convenient **rec** syntax we discussed in §4.2.4. This loop combinator can be considered a variant of the recursion combinator proposed for the original Lava (§4.2.4), or a generalisation of the original μ operator of μFP (§4.2.1). Intuitively loop f allows f to recursively define and use a value of type γ, but the effects of f should only be done once.

To support combinationally-cyclic circuits we define the ArrowCombLoop class:

$$\begin{array}{l} \textbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow \mathsf{ArrowCombLoop}\ (\rightsquigarrow)\ \gamma\ \textbf{where} \\ \quad \mathsf{combLoop} :: ((\alpha, \gamma) \rightsquigarrow (\beta, \gamma)) \to (\alpha \rightsquigarrow \beta) \end{array}$$

In contrast to loop we cannot expect combLoop to be uniformly polymorphic as we typically use Kleene iteration to find the fixed points of these cycles, and so need a (reified!) least element to start from. Similarly we cannot expect combLoop to perform the effects of its argument once, and so we lose many of the properties of loop. In particular we cannot move impure Arrows out of the scope of combLoop (Paterson's TIGHTENING rules), but the remainder involving pure Arrows should be satisfied:

EXTENSION $\quad \mathsf{loop}\ (\mathsf{arr}\ f) = \mathsf{arr}\ (\mathsf{trace}\ f)$

SLIDING $\quad \mathsf{loop}\ (f \ggg \mathsf{arr}\ (id \times k)) = \mathsf{loop}\ (\mathsf{arr}\ (\mathsf{id} \times k) \ggg f)$

VANISHING $\quad \mathsf{loop}\ (\mathsf{loop}\ f) = \mathsf{loop}\ (\mathsf{arr}\ \mathsf{unassoc} \ggg f \ggg \mathsf{arr}\ \mathsf{assoc})$

SUPERPOSING $\quad \mathsf{second}\ (\mathsf{loop}\ f) = \mathsf{loop}\ (\mathsf{arr}\ \mathsf{assoc} \ggg \mathsf{second} f \ggg \mathsf{arr}\ \mathsf{unassoc})$

where

$$\begin{aligned}
&\mathsf{trace} :: ((\alpha, \gamma) \to (\beta, \gamma)) \to \alpha \to \beta \\
&\mathsf{trace} f = \lambda x.\ \mathbf{let}\ (y, z) = f\ (x, z)\ \mathbf{in}\ y
\end{aligned}$$

$$\begin{aligned}
&\mathsf{assoc} :: ((\alpha, \beta), \gamma) \to (\alpha, (\beta, \gamma)) \\
&\mathsf{assoc} \sim (\sim (x, y), z) = (x, (y, z)) \\
&\mathsf{unassoc} :: (\alpha, (\beta, \gamma)) \to ((\alpha, \beta), \gamma) \\
&\mathsf{unassoc} \sim (x, \sim (y, z)) = ((x, y), z)
\end{aligned}$$

5.2.5 Meta-circuits

When modelling scenarios it is often convenient to add primitives that are not realisable; for instance we wish to model coin flips and abstract from the decisions of the environment, and to examine the internal state of a circuit without changing its interface. The following classes provide these facilities.

Probes

It is not always desirable or easy to expose all the signals of interest at module interfaces; doing so may violate abstraction and complicate composition. We therefore we provide *probes* that give names to arbitrary collections of signals:

$$\begin{aligned}
&\mathbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow) \Rightarrow \mathsf{ArrowProbe}\ (\rightsquigarrow)\ \alpha\ \mathbf{where} \\
&\quad \mathsf{probeA} :: \mathsf{ProbeID} \to (\alpha \rightsquigarrow \alpha)
\end{aligned}$$

We expect the Arrow ($\rightsquigarrow$) to record these probes in a global scope; we provide no means to access them from within a standard circuit but will use them extensively in concert with our machinery for knowledge-based circuits and model checking in the next chapter. While this approach is non-compositional, it is sufficient for our purposes to require that all labels used in a circuit be mutually distinct.

Non-determinism

Some behaviours of an environment or agent's behaviour are non-deterministic; some are genuinely so, such as an agent flipping a coin, and others are simply underspecified. We provide a basic class for non-deterministic choice:

```
class Arrow (⇝) ⇒ ArrowNonDet (⇝) α where
  nondetA :: (α, α) ⇝ α
  nondetFairA :: (α, α) ⇝ α
```

From this we can easily define other constructs such as a non-deterministic bit:

```
nondetBitA = trueA ‘nondetAC‘ falseA
```

We make very sparing use of fairness when it eases correctness assertions.

As such choices are represented by state variables, we also include a nondetLatchAC combinator where nondetLatchAC p chooses a value that satisfies p in the first instant, and returns that forever more. Using delayA in this way would use twice as many state variables as necessary. Similarly nondetChooseAC p makes a fresh choice every instant.

5.2.6 Two examples

Using the mechanisms defined above, we can describe the twisted ring counter of §4.1 as follows:

```
trc :: (ArrowComb (⇝), ArrowDelay (⇝)) ⇒ () ⇝ (B(⇝), B(⇝), B(⇝))
trc = proc () →
  do rec x ← (| delayAC (falseA −≺ ())
                        (| andAC (| orAC (returnA −≺ x) (notA −≺ y)) |)
                                 (notA −≺ z)) |) |)
         y ← (| delayAC (falseA −≺ ()) (returnA −≺ x) |)
         z ← (| delayAC (falseA −≺ ()) (returnA −≺ z) |)
     returnA −≺ (x, y, z)
```

We can use the **rec** syntax as all loops pass through a delay.

In contrast the cyclic circuit fgORgf of Figure 4.4 requires an explicit use of combLoop:

```
fgORgf :: (ArrowCombLoop (⇝) α, ArrowMux (⇝) α)
       ⇒ (α ⇝ α) → (α ⇝ α) → (B(⇝), α) ⇝ α
f ‘fgORgf‘ g = proc (choose, x) →
      (| combLoop (λgOut.
             do fOut ← f ⋘ muxA −≺ (choose, (x, gOut))
                gOut′ ← g ⋘ muxA −≺ (choose, (fOut, x))
                out ← muxA −≺ (choose, (gOut, fOut))
                returnA −≺ (out, gOut′) ) |)
```

```
counter :: Signal Bool → [Signal Bool]
counter inc = sum : sums
  where
    (sum, carryOut) = halfAdd (inc, sum′)
    sums = counter carryOut
    sum′ = delay False sum
```

```
halfAdd :: (Signal Bool, Signal Bool)
        → (Signal Bool, Signal Bool)
halfAdd (a, b) = (sum, carry)
  where
    sum   = xor a b
    carry = and2 a b
```

Figure 5.2: A counter with a bit width specified by its context.

We can also write it without the command combinator syntax:

```
f ‘fgORgf‘ g = combLoop arrow
  where
    arrow = proc ((choose, x), gOut) →
      do fOut ← f ⋘ muxA −≺ (choose, (x, gOut))
         gOut′ ← g ⋘ muxA −≺ (choose, (fOut, x))
         out ← muxA −≺ (choose, (gOut′, fOut))
         returnA −≺ (out, gOut′)
```

5.3 Datatypes and the need for generics

Up to this point data in our circuits has consisted of Booleans structured with tuples. The examples of the next chapter make use of more complex types such as numbers, and as these types must be finite in extent we often wish to parametrise them by their size.

To motivate our design decisions, consider the counter circuit due to Claessen (2001, p11) shown in Figure 5.2, expressed in the syntax of Lava 2000. This describes a binary counter that is incremented in every instant that its input is true, with the bit width specified by the circuit connected to its output. Claessen asserts that this circuit has “conceptually [...] infinite size.”

Our Arrow setting forces us to treat circuits (Arrows) and circuit generators (Haskell functions that return Arrows) separately. Moreover, as several of our instances of these classes (see §5.4) “statically” analyse the circuit’s graph in a similar way to the parser Arrows of §5.1.2, we cannot assume that the (⋙) operation is always lazy enough to construct these “infinite” circuits.

Therefore we would render counter as a generator parametrised by the output bit width:

```
counterAn :: (ArrowComb (⇝), ArrowDelay (⇝) (B(⇝)), ArrowLoop (⇝))
          ⇒ Integer → (B(⇝) ⇝ [B(⇝)])
counterAn 0 = arr (λ_. [])
counterAn n = proc carryIn →
  do rec (sum, carryOut) ← halfAddA −≺ (carryIn, sum′)
         sums ← counterAn (n − 1) −≺ carryOut
         sum′ ← (| delayAC (falseA −≺ ()) (returnA −≺ sum) |)
     returnA −≺ sum : sums
```

Unfortunately this does not work well in concert with our type-class based approach: in general we cannot provide an instance for ArrowDelay at type $[\alpha]$ as we have no way of communicating the list's length to delayAC. Moreover for types of arbitrary shape (e.g., trees) we would also need to indicate what this bound means.

We resolve this problem by including sizes in types. Clearly all non-recursive Haskell types already do this, but it can be quite complex for general algebraic datatypes. It suffices for our purposes to treat *sized lists*, which can be described by their carrier type and a type-level natural, a *phantom type*:

```
newtype SizedList size α = SizedList [α]
```

We hide the SizedList constructor from the end user so that we can enforce the invariant that the list is in fact of length *size*. The exported constructor mkSizedListA checks that the list length and type coinicide. We index SizedLists from 1 to $size$. We provide combinators for this type that parallel those in the Haskell Prelude for lists.

Somewhat arbitrarily, we use English cardinals to name the types we use as *size*s: One, Two, and so forth. We ask that all such types belong to the Card class:

```
class Card α where c2num :: Num n ⇒ α → n
```

We would therefore define a contextually-sized counterA as:

```
counterA :: forall (⇝) size.
                (ArrowComb (⇝), ArrowDelay (⇝) (B(⇝)), ArrowLoop (⇝), Card size)
            ⇒ B(⇝) ⇝ SizedList size B(⇝))
counterA = counterAn (c2num (undefined :: size)) ⋙ mkSizedListA
```

The undefined noise is the idiomatic way of translating sizes encoded in types into Integers: the explicit **forall** brings the type variables into scope in the definition.

We also provide CardAdd c_1 c_2 and CardMul c_1 c_2 types with instances of the Card class that perform the corresponding arithmetic operations. Our ambitions in this direction are intentionally limited as we expect the Glasgow Haskell Compiler to have much better support for type-level natural numbers in the near future.

In addition we also use some very simple-minded functorial generic classes that witness an isomorphism between a structured type α and a list of its components $[\delta]$. We can *destructure* any type:

```
class StructureDest δ α where
  destructure :: α → [δ]
```

In practice we expect the list to be finite.

For types that reflect the size of their inhabitants we can also map $[\delta]$ back into an α:

$$\begin{array}{l}
\textbf{class}\ (\mathsf{StructureDest}\ \delta\ \alpha,\ \mathsf{Card}\ (\mathsf{SIWidth}\ \delta\ \alpha))\ \Rightarrow\ \mathsf{Structure}\ \delta\ \alpha\ \textbf{where} \\
\quad \textbf{type}\ \mathsf{SIWidth}\ \delta\ \alpha\ ::\ * \\
\quad \mathsf{structure}\ ::\ \mathsf{StateM}\ [\delta]\ \alpha
\end{array}$$

The type function SIWidth gives the length of the list $[\delta]$ as a type.

We avoid using the Functor class simply so that we can use our base types (such as $\mathsf{B}(\rightsquigarrow)$) generically (e.g., with delayA and muxA) without having to put them in a trivial container. The classes are split as we will sometimes need to destructure recursive types; see §6.1.

5.3.1 Sized saturated natural numbers

Using our type class pattern we can easily specify the signature of the basic ordering combinators similarly to the Haskell Prelude:

$$\begin{array}{l}
\textbf{class}\ \mathsf{ArrowComb}\ (\rightsquigarrow)\ \Rightarrow\ \mathsf{ArrowEq}\ (\rightsquigarrow)\ \alpha\ \textbf{where} \\
\quad \mathsf{eqA}\ ::\ (\alpha,\ \alpha)\ \rightsquigarrow\ \mathsf{B}(\rightsquigarrow)
\end{array}$$

$$\begin{array}{l}
\textbf{class}\ \mathsf{ArrowEq}\ (\rightsquigarrow)\ \alpha\ \Rightarrow\ \mathsf{ArrowOrd}\ (\rightsquigarrow)\ \alpha\ \textbf{where} \\
\quad \mathsf{leA}\ ::\ (\alpha,\ \alpha)\ \rightsquigarrow\ \mathsf{B}(\rightsquigarrow) \\
\quad \mathsf{ltA}\ ::\ (\alpha,\ \alpha)\ \rightsquigarrow\ \mathsf{B}(\rightsquigarrow)
\end{array}$$

There is a subtlety in defining the arithmetic class however, regarding multiplication. The obvious signature:

$$\mathsf{ArrowNum}\ (\rightsquigarrow)\ n\ \Rightarrow\ \mathsf{mulA}\ ::\ (n,\ n)\ \rightsquigarrow\ n$$

disregards the fact that the result of a multiplication has twice the width of its inputs. We could ask that the user pad out the arguments, but this is inefficient (§6.4.2), and so we have the instance specify the output type as a function of the input type:

$$\begin{array}{l}
\textbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow)\ \Rightarrow\ \mathsf{ArrowNum}\ (\rightsquigarrow)\ n\ \textbf{where} \\
\quad \textbf{type}\ \mathsf{MulOut}\ (\rightsquigarrow)\ n\ ::\ * \\
\quad \mathsf{addA}\ ::\ (n,\ n)\ \rightsquigarrow\ n \\
\quad \mathsf{subA}\ ::\ (n,\ n)\ \rightsquigarrow\ n \\
\quad \mathsf{mulA}\ ::\ (n,\ n)\ \rightsquigarrow\ n \\
\quad \mathsf{fromIntegerA}\ ::\ \mathsf{Integer}\ \rightarrow\ (\gamma\ \rightsquigarrow\ n)
\end{array}$$

The type of arithmetic we use most in our modelling is over the *saturated naturals*, which is represented by some set isomorphic to $\{0..2^{size} - 1\}$ where all operations truncate at both ends. We define the carrier types as a *data type family*:

$$\textbf{data family}\ \mathsf{Nat}\ ((\rightsquigarrow)\ ::\ *\ \rightarrow\ *\ \rightarrow\ *)\ (size\ ::\ *)\ ::\ *$$

This declares a type constructor Nat which takes a pair of types – an Arrow ($\rightsquigarrow$) and a bit width *size* – into a type that represents the saturated naturals of that width for that Arrow. (Type- and data families are the stand-alone generalisation of the *associated types* we used in the ArrowComb class in §5.2.1 and MulOut above.) This decoupling of types from representations is one of the original motivations for type families (Chakravarty et al. 2005).

Following the idioms of the preceding section, we can define a set of circuit generators by recursion over the bit width and then use our sized types to provide a friendlier abstraction. For instance, we can declare an instance of the Nat data type family for the constructivity Arrow CArrow we will meet in §5.4.3 by deciding on a bit-level representation:

```
data instance Nat CArrow w = NatCArrow [B CArrow]
unNatCArrow (NatCArrow n) = n
```

Using the equalA circuit generator:

```
equalA :: ArrowComb (⇝) ⇒ Integer → (([B(⇝)], [B(⇝)]) ⇝ B(⇝))
equalA n = zipWithA n iffA >>> conjoinA n
```

where zipWithA :: Arrow ($\rightsquigarrow$) $\Rightarrow$ Integer $\to$ ((α, β) $\rightsquigarrow$ δ) $\to$ (([α], [β]) $\rightsquigarrow$ [δ]) is the Arrow version of zipWith, we give an instance of the ArrowEq class for all bit widths w:

```
instance (Arrow (⇝), Card w) ⇒ ArrowEq CArrow (Nat CArrow w) where
  eqA = arr unNatCArrow *** arr unNatCArrow >>> equalA size
    where size = c2num (undefined :: w)
```

We repeat this pattern for the other arithmetic classes. The instances for the Structure and StructureDest classes are similarly straightforward. While the boilerplate is rather heavy, it could be generated automatically.

A benefit of the type family approach is that we can interpret arithmetic at different levels of abstraction: for instance, at the bit level, where the operations are constructed from gates, or at the architectural level using the arithmetic operations of the host Haskell system (see §5.4.1). This would be impossible if we directly mapped arithmetic to bit-level operations, as is typically done in the systems we surveyed in §4.2. We can also ameliorate the concern that our circuit interpretations give divergent meanings to these operations by sharing their implementations; for instance, a bit-level adder is generated by the same code for all of the interpretations.

We provide the natA and constNatA combinators to fix the arithmetic representations, which often obviates type signatures, and an operation numCastA for casting between widths.

5.3.2 Concluding Remarks

Defining arithmetic circuits as Lava 2000 does, by recursion on their input lists, requires the user to know that the output of a multiplier is twice as wide as its two inputs, and that circuits with

```
mapACn :: Arrow (⇝) ⇒ Integer → (Integer → ((γ, α) ⇝ δ)) → ((γ, [α]) ⇝ [δ])
mapACn n f = go 1
  where
    go i | i == n + 1 = proc (env, []) → returnA −≺ []
         | otherwise = proc (env, b : bs) → (| (liftA2 (:)) (f i −≺ (env, b))
                                                           (go (succ i) −≺ (env, bs)) |)
```

Figure 5.3: The general mapACn combinator fails to be a command combinator.

two inputs expect them to be the same width. Enforcing invariants such as these is the job of types. As we observed in §5.3, we expect this to be much easier to do in the near future.

The need for sized types for bit representations was observed by Claessen (2001, p11) and Diatchki et al. (2005), and has also been adopted by Gill and Farmer (2011) in the context of circuits. We refrain from further discussion of the vast field of generic programming.

There is one more subtlety in our use of command combinators such as zipWithA: we often want to parametrise the argument Arrow by list index. Take, for example, our most-general list map function shown in Figure 5.3. This fails to be a command combinator as the argument is not an Arrow, and therefore we cannot reuse the general Arrow plumbing. This further motivates putting sizes in types. It also implies that we need several map-like operators for use in command combinator settings.

5.4 Interpretations of Circuit Descriptions

Our basic circuit interpretations are netlists (static structure) and simulation (dynamic semantics). We also construct symbolic representations suitable for state-space traversal.

5.4.1 Netlists

The simplest interpretation involves translating a circuit description into a netlist, which is simply a graph representing a traditional schematic diagram. As we do no further processing of these graphs we adopt a simple representation using association lists:

```
newtype NodeID = NodeID Int
type AssocList = [(NodeID, [Wire])]
type NetList = (Nodes, AssocList)
```

Each circuit component is given a unique NodeID, and Nodes collects their descriptions. The type Wire consists of an origin NodeID and a description.

We use the following Arrow:

```
newtype NLArrow detail α β = NLArrow (α → StateM [NodeID] (NetList, β))
```

where StateM is the (pure) state Monad we discussed in §5.1.2.

Instances for the standard Arrow classes and those introduced in §5.2 are straightforward, using the intuition mentioned in §5.1.2 that wires carry the NodeID of the circuit that drives them. Methods for components also add entries to the graph. The representation of the Boolean type B(NLArrow *detail*) in the ArrowComb class is NodeID.

The run function has the signature:

$$\begin{aligned}\mathsf{runNL} &:: (\mathsf{Structure\ NodeID}\ \alpha,\ \mathsf{StructureDest\ NodeID}\ \beta)\\ &\quad \Rightarrow \mathsf{NLArrow}\ \mathit{detail}\ \alpha\ \beta \rightarrow \mathsf{NetList}\end{aligned}$$

We generate an input for the Arrow using the methods provided by Structure NodeID α, and capture its output using StructureDest NodeID β.

The *detail* phantom type parameter allows us to provide either a single instance for all levels of detail, or separate ones. For example, we provide a single interpretation of the ArrowComb class as it contains no substructure. In contrast the sized arithmetic of §5.3.1 is given two meanings: at the *architectural* level we leave the implementation of the operations opaque, so they appear as boxes in the netlist, and at the *implementation* level we show their definition in terms of bits.

We illustrate this effect with an implementation of Euclid's algorithm for computing the greatest common divisor of two positive integers. The Arrow is shown in Figure 5.4, with its netlists at the two levels shown in Figures 5.5 and 5.6.

The netlist interpretation of the fgORgf circuit generator example of §5.2.6 is shown in Figure 5.7. As we can only interpret circuits, not circuit generators, we have applied fgORgf to circuits f and g that only have netlist semantics.

This interpretation was in essence presented by Erkök (2002, §1.2) in a Monad context, and Paterson (2003) for Arrows. An interpretation that translates our circuit descriptions into other languages as VHDL or Verilog could use this technique. It is straightforwardly extended with geometrical structure along the lines of μFP(§4.2.1), as we hint at by identifying sub-circuits using note. Further development of such combinators is beyond the scope of this project.

5.4.2 Simulation

Our simulation instance could be based on lazy lists, taking care with combinational loops (§4.1). We instead define our SyncFun Arrow transformer using the coiterative representation developed by Caspi and Pouzet (1998):

$$\mathbf{data}\ \mathsf{SyncFun}\ \mathit{detail}\ (\rightsquigarrow)\ \alpha\ \beta\ =\ \mathbf{forall}\ s.\ \mathsf{SyncFun}\ ((\mathsf{Bool},\ s,\ \alpha)\ \rightsquigarrow\ (s,\ \beta))$$

As for netlists we support bit and architecture level simulations with the *detail* parameter. The Boolean is true only in the initial instant. The underlying Arrow ($\rightsquigarrow$) is typically either the pure function Arrow ($\rightarrow$), or the Kleisli lifting of the IO Monad $\alpha \rightarrow \mathsf{IO}\ \beta$. The existentially-quantified

```
gcd :: (ArrowDelay (⇝) (n, n), ArrowLoop (⇝), ArrowMux (⇝) (n, n),
        ArrowNum (⇝) n, ArrowOrd (⇝) n) ⇒ (B(⇝), (n, n)) ⇝ (B(⇝), n)
gcd = proc inputs → do xy ← note "Computation" comp −≺ inputs
                       eqA &&& arr fst −≺ xy
  where
    comp = proc (input_ready, xy0) →
      do rec xy@(x, y) ← (| muxAC (returnA −≺ input_ready)
                                  (returnA −≺ xy0)
                                  (| delayAC (returnA −≺ xy0)
                                             (returnA −≺ xy′) |) |)
             xLEy ← leA −≺ xy
             xSUBy ← subA −≺ xy
             ySUBx ← subA ⋘ swapA −≺ xy
             xy′ ← muxA −≺ (xLEy, ((x, ySUBx), (xSUBy, y)))
         returnA −≺ xy
```

Figure 5.4: A hardware implementation of Euclid's algorithm for integers of width n.

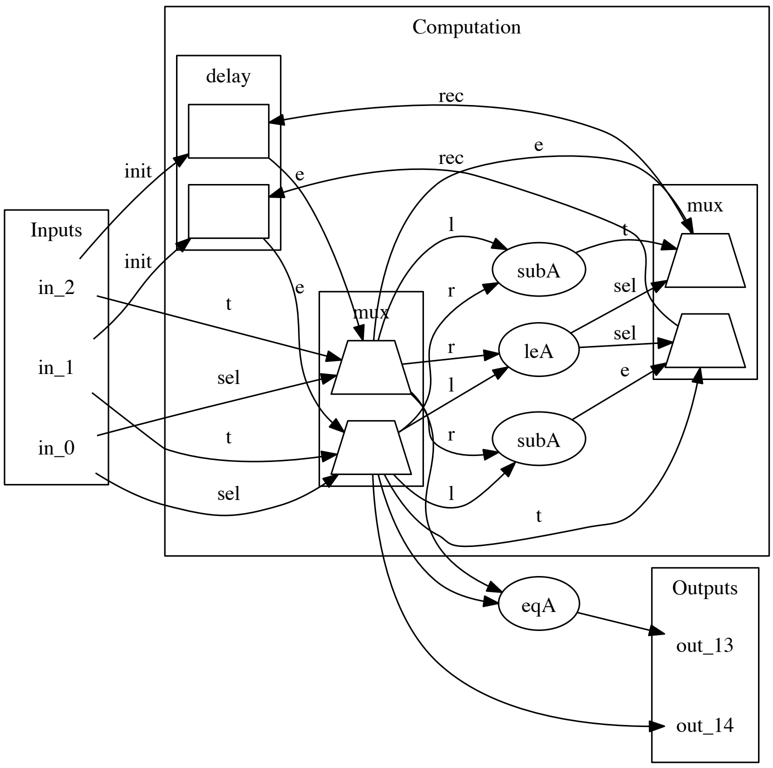


Figure 5.5: The architecture-level netlist of the GCD circuit, rendered using Graphviz.

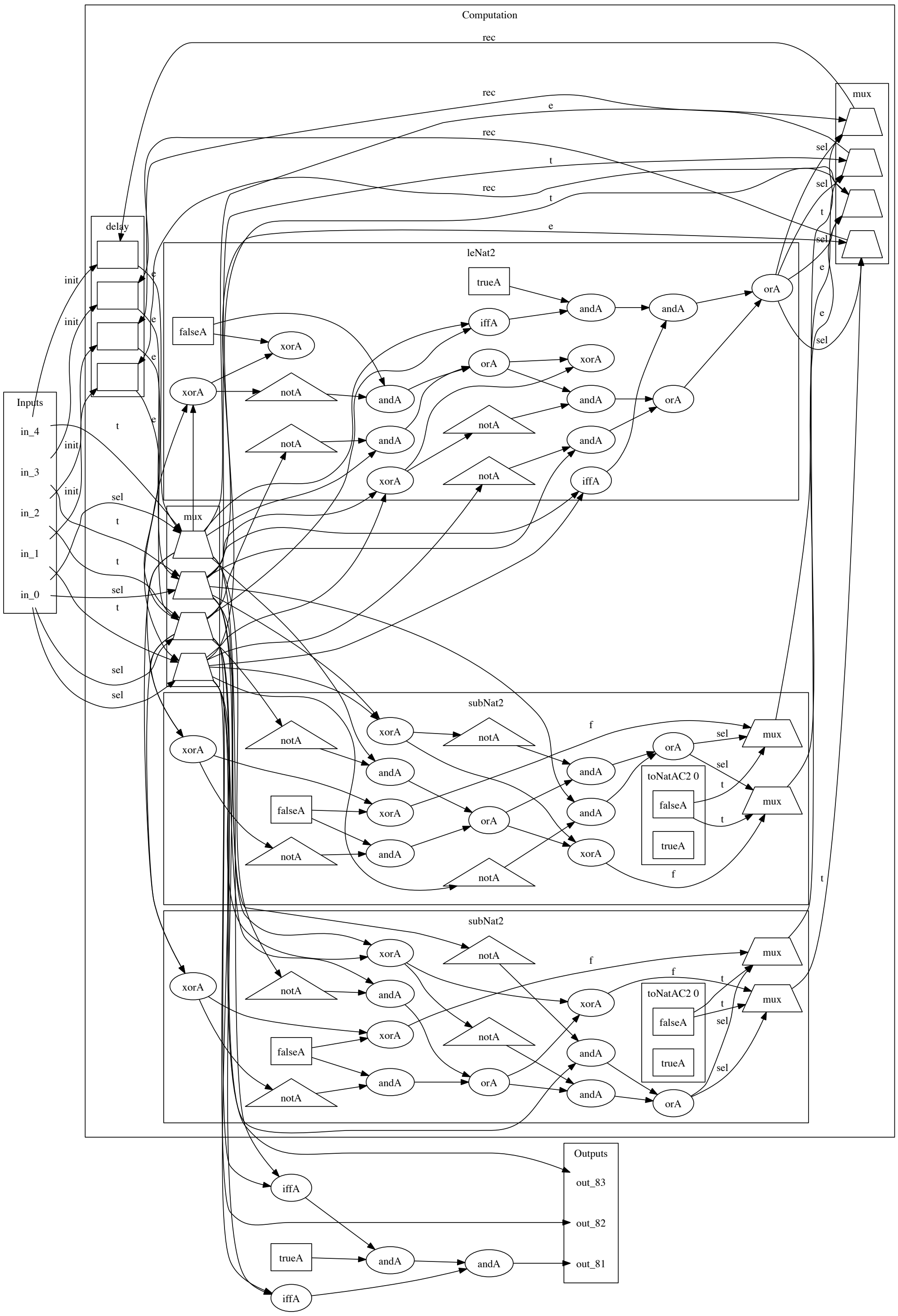


Figure 5.6: The implementation-level netlist of the GCD circuit assuming a bit-width of two, rendered using Graphviz.

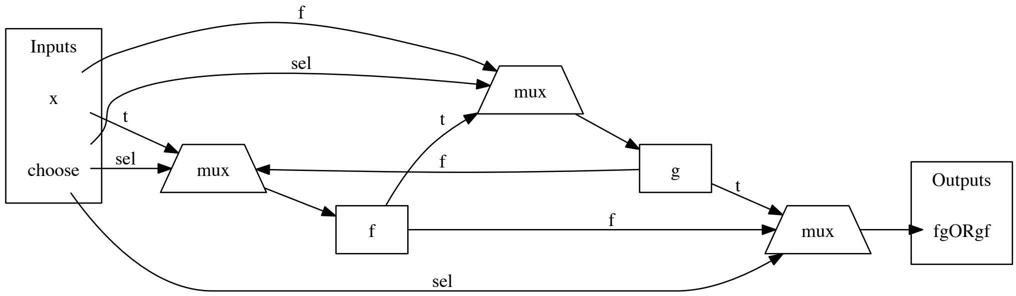


Figure 5.7: The netlist of the running fgORgf example, rendered using Graphviz.

type variable s represents the state of the circuit. The state for Arrows constructed using the ($\ggg$) and first combinators is the pair of the states of their argument Arrows. Initially the state is globally undefined ($\bot$), and the $\ggg$ and first operations lazily split the incoming state between their arguments, and combine the new state from their sub-Arrows.

To support the Kleene iteration of the combinational cycles we provide the SimBool type that explicitly represents the domain of §4.1, along with the corresponding operations. The ArrowCombLoop instance exploits the explicit representation of the temporal state by holding it constant while computing the fixed point of its argument.

This representation is amenable to implementation in a low-level imperative language, and could form the core of an explicit-state model checker.

5.4.3 Constructivity Analysis

As we will see in Chapter 6, our Haskell implementation of the algorithms of §3.6.7 depends on the transition relation of the environment being encoded as a Boolean decision diagram (BDD), a data structure that we discussed in §2.3.2. Therefore we seek to transform combinationally-cyclic sequential circuits into standard classical circuits that we can represent in the traditional manner. We do this by adapting the algorithm of Shiple et al. (1996) to our Arrow setting.

We denote the three-valued Boolean domain of §4.1 by $\mathbb{B}$. Shiple et al. (1996) represent this domain using the two-valued Boolean type $\mathbb{T}$ by employing a *dual-rail encoding*, where a function $f :: \mathbb{B}^n \to \mathbb{B}$ is represented by a pair of functions $(f^F, f^T) :: ((\mathbb{T} \times \mathbb{T})^n \to \mathbb{T})^2$. The function $f^F\,\vec{x}$ yields true iff $f\,\vec{x}$ is F, and $f^T\,\vec{x}$ is true iff $f\vec{x}$ is T, where we appeal to the embedding of $\mathbb{B}$ into $\mathbb{T} \times \mathbb{T}$ given by mapping T to (false, true), F to (true, false) and the divergent behaviour $\bot$ to (false, false). The "top" value (true, true) is unused. The basic gates are defined as follows:

$$\mathsf{notA}\ (x^F, x^T) = (x^T, x^F)$$
$$\mathsf{andA}\ ((x^F, x^T), (y^F, y^T)) = (x^F \vee y^F, x^T \wedge y^T)$$

Note that the rails are entangled by the notA operation. The key property of this particular encoding is that a function classically equivalent to f is given by f^T provided that f is constructive, i.e. always defined.

Their algorithm processes combinational loops using a nested fixed-point computation. To efficiently compute these we need to preserve the state of inner fixed-point computations between iterations, which leads us to define a "dynamic" Arrow that interprets the circuit itself:

$$\textbf{newtype}\ \mathsf{TwoRails}\ \alpha\ =\ \mathsf{TwoRails}\ (\alpha,\ \alpha)$$
$$\textbf{newtype}\ \mathsf{DynArr}\ \alpha\ \beta\ =\ \mathsf{DynArr}\ (\mathsf{StateArrow}\ \mathsf{Dynamic}\ (\rightarrow)\ (\mathsf{TwoRails}\ \alpha)\ (\mathsf{TwoRails}\ \beta))$$

The Dynamic type is a record containing the fixed-point state, a description of the initial states and transition relation, and the miscellany required by the machinery for knowledge (§6.1). The Arrow transformer StateArrow adds state to the underlying Arrow ($\rightsquigarrow$):

$$\textbf{newtype}\ \mathsf{StateArrow}\ s\ (\rightsquigarrow)\ \alpha\ \beta\ =\ \mathsf{StateArrow}\ ((\alpha,\ s)\ \rightsquigarrow\ (\beta,\ s))$$

We observe that the DynArr type is isomorphic to a Kleisli Arrow using the state Monad:

$$\mathsf{DynArr}\ \alpha\ \beta\ \simeq\ \mathsf{TwoRails}\ \alpha\ \rightarrow\ \mathsf{StateM}\ \mathsf{Dynamic}\ (\mathsf{TwoRails}\ \beta)$$

Instantiating the Arrow and ArrowLoop classes for DynArr is straightforward, noting that we need to explicitly mediate the isomorphism:

$$\mathsf{TwoRails}\ ((x^F,\ y^F),\ (x^T,\ y^T))\ \simeq\ (\mathsf{TwoRails}\ (x^F,\ x^T),\ \mathsf{TwoRails}\ (y^F,\ y^T))$$

The ArrowComb instance defines the associated type B to be CBool, a renaming of the BDD type that has no instances. This guarantees that the end user cannot manipulate the representation using pure Arrows. The methods of the class are defined with the two-rail Boolean functions sketched above. The remaining classes require some context, which we combine with a treatment of sequential circuits.

Shiple et al. (1996) lift this analysis to sequential circuits by identifying the states where there is a wire that has an undefined value, and determining if these states are reachable. As per tradition we use pairs of BDD variables to represent the relation between the present state and the next in a transition system. There is no need for a dual-rail representation of the state itself as it will always be well-defined provided the combinational part of the circuit is constructive; our sequential constructivity analysis ensures this invariant.

We allocate these BDD variables "statically", i.e., once-and-for-all, we use a state Monad to track this information. Thus we define our constructivity analysis Arrow:

$$\textbf{newtype}\ \mathsf{CArrow}\ \alpha\ \beta\ =\ \mathsf{CArrow}\ (\mathsf{StateM}\ \mathsf{Static}\ (\mathsf{DynArr}\ \alpha\ \beta))$$

Again, the Arrow and ArrowLoop instances are standard, and the ArrowComb instance is a simple lifting of that for DynArr. We construct the ArrowMux instance from ArrowComb and the generics of §5.3. The ArrowDelay instance makes use of the BDD variable allocation recorded in Static. Non-determinism (§5.2.5) is encoded in the state as extra BDD variables which are suitably constrained.

The key ArrowCombLoop instance computes the fixed point of combinational cycles locally, using the strategy of Bourdoncle (1993).

The reader might compare the type given for CArrow with that given for Hughes's parsers in §5.1.2; both are designed to split a computation into static and dynamic parts. Whereas Hughes appeals directly to the monoids underlying his analysis, we thread Static data throughout the circuit and do not assume that it can be combined monoidally. We also note that CArrow is not a Kleisli Arrow; it does not support a Monad instance.

The problem of verifying that a circuit containing combinational cycles is well-defined has been treated at length in the literature. Malik (1993) treated cyclic combinational circuits. Shiple et al. (1996) extended his work to combinationally-cyclic sequential circuits and related this ternary analysis to the physical circuit models of Brzozowski and Seger (1995). Namjoshi and Kurshan (1999) observe that we can use any fixed point of the circuit equations to determine constructivity, and encode the problem in SAT. Their method does not yield an equivalent combinationally-acyclic circuit however. Claessen (2003) augments their approach with temporal induction to prove safety properties directly on the cyclic circuit. Further context can be found in Neiroukh et al. (2008).

We are not too concerned about optimising our constructivity analysis as its runtime is dominated by the construction of the knowledge automata (§6.2).

5.5 Kesterel: Esterel as an Arrow Transformer

While circuits provide a convenient way of describing dataflow computations, many KBPs have an imperative flavour and are better expressed as state machines. We would like to find a set of combinators that work well on this domain.

For small designs it is typically easy to explicitly spell out the state machine, but such descriptions are not compositional; in other words, small changes in desired behaviour may require large changes in structure (Berry 1999b, §3.1.3). Moreover the class of deterministic reactive state machines (i.e., those that can be mapped to circuits) is not closed under synchronous composition (Maraninchi and Halbwachs 1996). For these reasons we seek to adapt the mature imperative synchronous language Esterel (Potop-Butucaru et al. 2007) to our Arrow setting, as it addresses these issues; indeed, it substantiates Berry's claim (§4.3.1) that synchronous languages have reconciled determinism and concurrency.

The following sections sketch the main features of Esterel and our implementation of its circuit semantics as an Arrow transformer, which we call Kesterel as we will soon add knowledge to it. Our goal is to develop a framework for language experimentation; efficiency is not the primary consideration. We use it to describe cache protocols in §6.6. We also use a much simpler embedded imperative language in §6.4.2.

5.5.1 The Esterel Language

Esterel has been canvassed at length in the literature; see Berry (1999a,b); Potop-Butucaru et al. (2007) amongst many others. Here we content ourselves with a brief overview.

The language includes a battery of familiar imperative constructs: variable assignment, conditionals, loops, exception handling, sequential composition. To these are added the key ingredient for supporting synchrony: the `pause` statement, which has the effect of halting a thread of control for an instant when it is active. "Compile-time" concurrency is provided by parallel composition, and prioritised preemption allows computations to be modularly aborted. There is also a notion of suspension, which inactivates a component under some condition. All of these are carefully constructed to preserve determinism and reactivity.

Communication amongst threads is by instantaneous signal broadcast within some scope, reminiscent of a wire in a circuit: at each instant, each signal is either present or absent.

Berry (1999b, §3.1) gives this simple example specification and Esterel implementation:

`ABRO`: Emit an output `O` as soon as two inputs `A` and `B` have occurred. Reset this behavior each time the input `R` occurs.

```
module ABRO:
input A, B, R;
output O;
loop
  [ await A || await B ];
  emit O
each R
end module
```

Intuitively the parallel composition `[ await A || await B ]` implements the condition in the first part of the specification, with `emit O` generating the required output. The reset behaviour is handled by the preemptive loop construct `loop ... each R`. Many such special forms are provided to handle the timing subtleties that arise in practice; see Berry (1999b) for details. Berry argues that this implementation scales easily and linearly with the number of inputs it waits for, whereas the corresponding state machine requires extensive reworking.

Esterel has been given a variety of semantics over its history. Here we use the simple translation to cyclic circuits given by (Berry 1999a). As we discussed in §4.1, this grounds the semantics of Esterel in physical electrical models. Intuitively cycles are used to broadcast signal statuses, and constructivity ensures that the resulting circuit is logically well-defined.

This translation demonstrates how Esterel resolves parallel compositions at compile-time, i.e. how there need not be any concurrent activity at runtime.

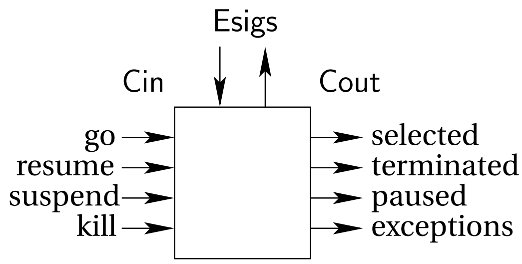


Figure 5.8: The interface to the circuit representing an Esterel statement. See Berry (1999a, §11.2.1) for details.

5.5.2 Implementation as an Arrow Transformer

The translation given by Berry (1999a) maps Esterel expressions into circuits with the interface shown in Figure 5.8. We use a pair of records to model the control inputs Cin and Cout, and sequences of Booleans to track the signal environment ESigs and thrown exceptions.

We structure our translation as a shallow embedding using the familiar static/dynamic split. In this instance the static information is the translation context, which is just the number of signals and exceptions in scope. The generated dynamic Arrow incorporates the standard circuit interface for Esterel constructs:

$$\begin{array}{l}\mathbf{type}\ \mathsf{Dynamic}\ (\rightsquigarrow)\ \alpha\ \beta\ =\ (\mathsf{Cin}\ (\mathsf{B}(\rightsquigarrow)),\ \mathsf{Esigs}\ (\mathsf{B}(\rightsquigarrow)),\ \alpha)\ \rightsquigarrow\ (\mathsf{Cout}\ (\mathsf{B}(\rightsquigarrow)),\ \mathsf{Esigs}\ (\mathsf{B}(\rightsquigarrow)),\ \beta)\\ \mathbf{newtype}\ \mathsf{E}\ (\rightsquigarrow)\ \alpha\ \beta\ =\ \mathsf{E}\ (\mathsf{EnvM}\ \mathsf{Static}\ (\mathsf{Dynamic}\ (\rightsquigarrow)\ \alpha\ \beta))\end{array}$$

Here we use a simple environment Monad EnvM:

$$\mathbf{newtype}\ \mathsf{EnvM}\ s\ \alpha\ =\ \mathsf{EnvM}\ (s \to \alpha)$$

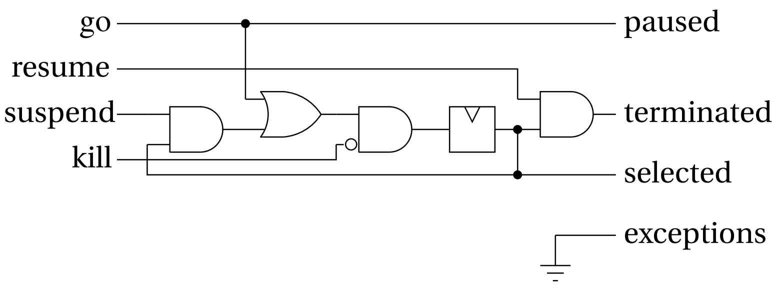
with an operation to read from the environment ($\mathsf{readEnvM} :: \mathsf{EnvM}\ s\ s$) and another to run a computation in a new environment ($\mathsf{inEnvM} :: s \to \mathsf{EnvM}\ s\ \alpha \to \mathsf{EnvM}\ s\ \alpha\ ()$).

It is straightforward to implement the Arrow and ArrowLoop classes using the semantics for Esterel's sequential composition. We provide an instance for the ArrowTransformer class that lifts computations in the underlying Arrow $(\rightsquigarrow)$ into $\mathsf{E}\ (\rightsquigarrow)$ as instantaneous computations. Similarly the Esterel kernel statements are routine; for instance the circuit and E computation for the pauseE statement are shown in Figure 5.9.

We represent signals and exceptions as abstract indices into the environment, and provide allocation functions in the style of higher-order abstract syntax (HOAS):

$$\begin{array}{l}\mathsf{signalE} :: (\mathsf{EC}\ (\rightsquigarrow),\ \mathsf{Structure\ Signal}\ \upsilon) \Rightarrow (\upsilon \to \mathsf{E}\ (\rightsquigarrow)\ \gamma\ \alpha) \to \mathsf{E}\ (\rightsquigarrow)\ \gamma\ \alpha\\ \mathsf{catchE} :: \mathsf{EC} \Rightarrow (\mathsf{Exception} \to \mathsf{E}\ (\rightsquigarrow)\ \gamma,\ \alpha) \to \mathsf{E}\ (\rightsquigarrow)\ \gamma\ \alpha\end{array}$$

where EC collects the various circuit classes we discussed in §5.2. The generics of §5.3 allow for the allocation of any structure that can be constructed from Signals. The semantics of Esterel signals depends crucially on combinational cycles.




```
class (ArrowLoop (⇝), ArrowDelay (⇝) (B(⇝)),
        ArrowCombLoop (⇝) (B(⇝)), ArrowComb (⇝))
        ⇒ EC (⇝)
```

```
pauseE :: EC (⇝) ⇒ E (⇝) γ ()
pauseE = E ( do s ← readEnvM
                  return (proc (cin, ienv, c) →
                    do nKill ← notA −≺ ciKill cin
                       rec reg ← (| delayAC (falseA −≺ ()) (andA −≺ (t, nKill)) |)
                           t ← orA ⋘ second andA −≺ (ciGo cin, (ciSusp cin, reg))
                       terminated ← andA −≺ (reg, ciRes cin)
                       ff ← falseA −≺ ()
                       let (oenv, exns) = cenv_exns_empty s ff
                       returnA −≺ ( Cout { coSelected = reg
                                         , coTerminated = terminated
                                         , coPaused = ciGo cin
                                         , coExns = exns }, oenv, () ) ))
```

Figure 5.9: The circuit corresponding to a pauseE statement: Figure 11.3 from Berry (1999a) and its rendition as an Arrow. The function cenv_exns_empty yields exception and signal environments where all elements are set to absent.

Pleasantly the signal and exception scopes are enforced by the Arrow structure: Arrows may return Signals and Exceptions but there are no operations that accept them Arrow-bound. Hence we do not need to use type-level tricks to ensure that signals do not escape their scopes (in contrast to the ST Monad (Launchbury and Peyton Jones 1995), for instance).

The ABRO example of the previous section can be rendered as a Kesterel Arrow as follows:

```
abro a b r o = loopEachE r ( (awaitImmediateE a |||| awaitImmediateE b) ⋙ sustainE o )
```

where (||||) and (⋙) are parallel and sequential composition of E computations, respectively. The other operations use the standard expansions provided by Berry (1999a). We replace Esterel's module syntax with λ-bindings. This syntax is closer to the process calculus notation of Berry (1999a, Chapter 5) than traditional Esterel source code.

We note in passing that the netlist interpretation of §5.4.1 was very useful when debugging this Arrow transformer.

Kesterel is much easier to extend than existing Esterel implementations, which makes it an ideal platform for the sort of language experimentation we engage in in the next chapter. Moreover the standard Esterel language suffers from a lack of parametrisation in the same way as many standalone (non-embedded) DSLs; for instance signal routing can be quite verbose, at times dominating the control logic.

One of the major attractions for adding a Monadic interface to an EDSL is the **do** notation that idiomatically supports sequencing in Haskell. In our case the use of an Arrow transformer precludes a Monad instance, and unfortunately the Arrow syntax is heavier than explicit uses of the sequential composition operation ($\ggg$).

The simple translation we use here has issues with *reincarnation* and *schizophrenia*, which (coarsely put) describe the incorrect handling of some signals that are created in loop contexts. We provide an example in §6.6. A solution was proposed by Berry (1999a, Chapter 12), and all such solutions involve duplicating logic. As this severs the link between the source text and the generated circuit, it ceases to be meaningful to ask what the instantaneous value of a signal is – in general it may have as many statuses as the number of enclosing loops. In our case this complicates the identification of propositions in knowledge-based programs, and as resolving these issues is not critical to our agenda, we use the simple translation. We note that our Arrow-based scheme can support the full translation however, as it only involves adding context in ways we already support.

Similarly general data handling is complicated by parallel composition, and since our examples do not require it, we leave its addition to Kesterel to future work.

More broadly there has been much work towards efficient compilation of Esterel in software settings (Potop-Butucaru et al. 2007), in contrast to our goal of building a transition relation expressed as a BDD for exhaustive state-space traversal (§6.2). Bourke (2009, §2.4) discusses the hierarchical state machine language Argos and its relation to Esterel. Claessen (2001, Chapter 6) describes a simple imperative language called Flash that is compiled to combinationally cyclic circuits in Lava 2000 with some issues that are resolved by Esterel. York Lava, which we discussed in §4.2.6 provides a a small imperative language that does not treat the semantic complexities that Esterel does.

5.6 Concluding remarks

The Arrows presented here resolve the "observable sharing" issue we discussed in the previous chapter in a way that preserves the validity of the unrestricted β law in Haskell by introducing a combinatory language that is not Cartesian closed. We have also seen how other languages can be constructed on top of the basic circuit Arrows.

The approach sketched here relies on circuit descriptions being suitably polymorphic. We enforce this by only providing abstract interfaces to the basic components, which has the unfortunate effect that type signatures become quite verbose for any non-trivial circuit. As we tend to omit these, we need to defeat the standard Haskell monomorphism restriction as our circuits are overloaded constant applicative forms (CAFs) that are rejected by default if unaccompanied by a type signature.

In addition to the interpretations shown in §5.4, we would like to transform our circuit Arrows, such as by propagating constants. We leave this to future work.

Much of this chapter could also be carried out in a Monadic framework, or even using observable sharing; however the next chapter shows that our knowledge-based programs require one further construct that is not so readily implemented with these. We further evaluate the use of Arrows for this domain in §7.1.

Chapter 6

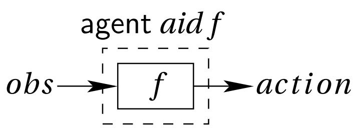
Knowledge-based circuits and applications

WITH the theory of Chapter 3 and machinery of Chapter 5 in hand, we return to the problem of constructing implementations of knowledge-based programs. We augment ADHOC with several constructs for describing knowledge-based programs, making essential use of the Arrow structure, and proceed to give symbolic versions of the algorithms we developed in Chapter 3 and canvas the problem of minimising the automata we generate. The remainder of the chapter illustrates the tools at work on a series of scenarios with epistemic characteristics.

6.1 Arrows for knowledge-based circuits

The Arrows of the previous chapter give us a compositional means of description that is easily parametrised, and for larger examples we have a flexible way of building state machines (§5.5). Our goal here is to extend them with constructs for knowledge-based programs.

We delimit the boundaries of an agent using the agent construct: §5.1.2:



class ArrowAgent $(\rightsquigarrow)$ *obs* **where**
  agent :: AgentID $\to$ $(obs \rightsquigarrow action) \to (obs \rightsquigarrow action)$

The environment can pass values of type *obs* to the agent, who responds with *action* values at each instant. Due to the Arrow structure these are the entirety of the "dynamic" interface between them. (The information passed "statically" to agents is commonly known to all agents.) As we need to capture the agents' observations but not their actions, we include only *obs* in the head of the ArrowAgent class; we use the generics of §5.3 to record the observation, as we show below. Agents can also maintain private state using delayA, which we can capture using the existing ArrowDelay class of §5.2.3. Similarly we capture the result of agent-local non-deterministic choices with the classes of §5.2.5, as these are represented by state variables.

Note that if we had used a Cartesian-closed abstraction such as observable sharing or Monads then we could not capture the observation in this way, as functions and Monadic computations cannot be scrutinised for what they depend upon. We discuss how agent boundaries could be enforced with these abstractions in §7.1.4. That this abstraction really does encapsulate an agent's state could probably be established by adapting the proof of non-interference by Li and Zdancewic (2010, §5) for their Arrow-based secure-computation EDSL, but the reader may be convinced by the examples in the following sections.

Our next step is to define a syntax for knowledge formulas analogous to the HOL datatype $('a, 'p)$ KForm of §3.2:

```
data KF = KFfalse | KFtrue | KF `KFand` KF | KFneg KF
        | KFprobe String    — propositions
        | AgentID `KFknows` KF
        | AgentID `KFknowsHat` ProbeID
        | [AgentID] `KFcommon` KF
        | [AgentID] `KFcommonHat` ProbeID
```

These constructors will not appear explicitly in the examples as we overload the common syntax for logical languages, using the familiar operators.

We deviate from our previous syntax in two substantive respects: firstly, our primitive propositions are circuit probes. Recall that interesting knowledge formulas refer to variables that are not in the agent's scope – and moreover in this setting an agent has direct knowledge of the values of all variables in its scope. By using probes we avoid the need to route unobservable values to agents, which would often severely obfuscate descriptions.

Secondly we add the modalities $\widehat{\mathsf{knows}}$ and $\widehat{\mathsf{cknows}}$ to make testing for an agent's knowledge of a variable more efficient. Semantically we expect:

$$\widehat{\mathsf{knows}}_a\ (v :: T) \equiv \bigvee_{i \in T} \mathsf{knows}_a\ (v = i)$$

where v is the representation of some probe, and i ranges over the elements of the type T. We expect a similar property of $\widehat{\mathsf{cknows}}$. This primitive is essential to our treatment of the Mr P. and Mr S. puzzle (§6.4.2) where T is large.

Using this syntax we define a construct for knowledge tests:

```
class ArrowKTest (⇝) where
  kTest :: KF → (γ ⇝ B(⇝))
```

In the scope of the agent method, kTest allows an agent to test the truth of the given knowledge formula, which is passed "statically"; these serve the same purpose as the guards in the Isabelle/HOL theory of Chapter 3. An agent may contain an arbitrary number of kTests; zero, in the case of model checking (§6.5), one (§6.3) or many (§6.4.2).

The crucial instances of these classes are for the constructivity Arrow CArrow of §5.4.3; we also lift this functionality to the Kesterel level in §6.6. The ArrowAgent instance captures the agent's observation of the environment using the generics of §5.3:

instance StructureDest CBool *obs* ⇒ ArrowAgent CArrow *obs* **where**
  agent *aid f* = ...

In other words, the environment can pass arbitrary (finite) structures to the agents, provided they are made out of bits. Agents' private states are similarly recorded by the ArrowDelay instance in the Dynamic structure. We note that StructureDest is sufficient here and allows us to give an instance for Kesterel, where signal environments are of arbitrary size.

The ArrowKTest instance for CArrow associates each KF formula in a kTest with a BDD variable, and stores these in the Static structure. Intuitively we compose the knowledge automaton in synchronous parallel with the rest of the system and use this bit to communicate the truth of the knowledge formula. We discuss this further in the next section.

We note that placing the agent method within a combinational cycle is difficult to interpret; essentially what the agent observes would depend instantaneously on what it does. There is the similar problem of allowing one agent to instantaneously observe the output of another's kTest. This can be resolved by adapting the constructivity analysis of §5.4.3 to order the kTests, and rejecting the program if this is not possible. As we make no use of combinational cycles involving the infrastructure for KBPs we do not pursue that here.

Conceivably the simulation Arrow of §5.4.2 could underpin an explicit-state variant. We leave this to future work.

6.2 Symbolic algorithms

We construct the automata representing kTests using a DFS as shown in Figure 3.4 on page 37. The equivalence classes of sets of system states (§3.6.4) are represented symbolically by *Boolean decision diagrams* (BDDs), which we discussed in §2.3.2. These provide canonical representations of Boolean functions, allowing the equality of two equivalence classes to be tested in $O(1)$ time. This is potentially more efficient than the ordered lists we used throughout §3.7 provided the BDDs are of tractable size.

The DFS also requires us to track the equivalence classes we have visited, and also to provide finite maps that represent the automata under construction. We do this explicitly as there is no obvious way to maintain sets of sets of system states symbolically without using exponentially many more BDD variables. We use the BDD handles (addresses) as keys in our automaton maps, and use a sparse representation – an association list – for the automata being constructed.

The pipeline for the construction process is shown in Figure 6.1. The circuit translation for Kesterel was discussed in §5.5, and the constructivity analysis in §5.4.3. We add the generated

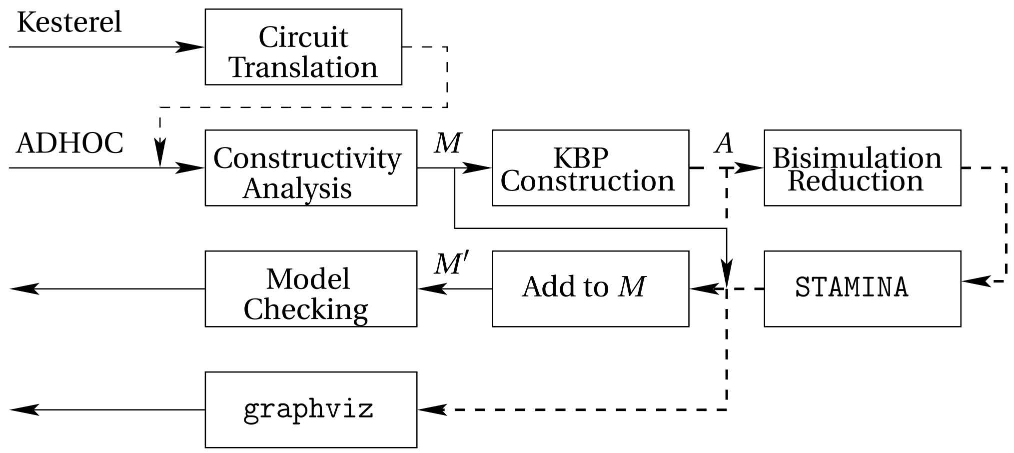


Figure 6.1: The pipeline for constructing implementations of KBPs. The symbolic model M is derived from an ADHOC or Kesterel description, and the automata A constructed using the algorithms of §6.2. The model M' is their composition. The minimisation steps are optional.

automata to the model by numbering their states and encoding their transition relations as BDDs, which we combine with the system's transition relation using a standard synchronous parallel composition (Clarke et al. 1999). This has the effect of defining the kTest BDD variables mentioned in §6.1.

The following sections describe the specific algorithms for constructing implementations of KBPs for each of the cases of §3.7. We continue the discussion we began in §3.8 about reducing the generated automata in §6.2.4, and conclude the chapter with a series of examples.

6.2.1 The Clock case

Intuitively the algorithm for the clock semantics of §3.7.1 can replace a pure depth-first search with a breadth-first search that proceeds by temporal slices.

Concretely we maintain the set of states reachable at time n and, for each agent a, partition these under a's observation function. We add states to the automaton for a for the new equivalence classes. For each new equivalence class ec, we add edges from *all* of the states in the previous temporal slice to ec, labelling them with the observation that a makes on ec. We can see that the resulting automaton is behaviourally equivalent (§3.6.2) to the one constructed by the algorithm in §3.7.1; we have simply added superfluous transitions.

After each equivalence class of the temporal slice has been processed, we construct the next slice using the standard idiom for BDDs; the evaluation function of §3.7.1 for knowledge formulas is easily adapted to use a symbolic representation.

This approach potentially saves time but not space as we compute the set of states commonly known to be possible only once per temporal slice.

6.2.2 The Single-Agent Perfect Recall case

The algorithm discussed in §3.7.3 for a single agent is readily translated into the present setting. We apply the implementation to the robot example in §6.3.

6.2.3 The Multi-Agent Broadcast Perfect Recall cases

The broadcast settings of §3.7 assume that the agents make common observations of the shared state, while allowing them to maintain their own private states. Therefore we introduce the ArrowBroadcast class so that we can capture this common observation and ensure that all communication between the agents is by broadcast:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{ArrowBroadcast}\ (\rightsquigarrow)\ \mathit{iobs}\ \mathit{cobs}\ \textbf{where} \\
\quad \mathsf{broadcast} :: \mathsf{Card}\ \mathit{size} \\
\qquad\qquad \Rightarrow \mathsf{SizedList}\ \mathit{size}\ (\mathsf{AgentID}, \mathit{ienv} \rightsquigarrow \mathit{iobs}, (\mathit{iobs}, \mathit{cobs}) \rightsquigarrow \mathit{action}) \\
\qquad\qquad \rightarrow (\mathit{env} \rightsquigarrow \mathit{ienv}) \\
\qquad\qquad \rightarrow (\mathit{env} \rightsquigarrow \mathit{cobs}) \\
\qquad\qquad \rightarrow (\mathit{env} \rightsquigarrow \mathsf{SizedList}\ \mathit{size}\ \mathit{action})
\end{array}
$$

Here *cobs* is the type of the common observation, *iobs* that of the agents' initial observations, and *ienv* the type of the initial environment from which the initial observations are made. (The initial environment allows computations to be shared.) The agents are presented as a SizedList (§5.3) of tuples, consisting of the agent's name, their initial observation and their recurring behaviour. The broadcast combinator returns an Arrow that maps the environment to a SizedList of actions, one per agent. This communicates to the environment's protocol that the number of actions is equal to the number of agents.

The instance of ArrowBroadcast for the constructivity analysis Arrow CArrow of §5.4.3 is similar to that for the agent construct we discussed in §6.1. The only subtlety is that we must provide the initial observation only in the initial instant; in particular we cannot give the agents' access to the output of the Arrow that generates this observation (the second in their defining tuple) at later instants.

Again the algorithms of §3.7 translate readily to this symbolic setting. We note that the representation of relations between the initial and present-state variables used in §3.7.4 and §3.7.5 involves another set of pairs of BDD variables in addition to the omnipresent past- and current-state ones that the previous algorithms have used.

6.2.4 Automata Minimisation

The automata generated by these processes contains much redundant structure, and as we wish to comprehend these artifacts we would like to find small behaviourally-equivalent automata (§3.6.2). We consider only schemes for the reduction of deterministic state machines here, and as our automata representations are explicit, we do not discuss symbolic techniques.

In §3.8 we used a standard DFA minimisation algorithm (Gries 1973) to reduce the size of our automata. A recent variant of this approach due to Valmari (2012) runs in $O(n \lg n)$ time where n is the number of states, independently of the size of the alphabet (observations in our case), and conveniently works on underspecified automata. However, as we remarked there, this standard reduction under bisimulation does not yield the smallest automata we could hope for as it is overly respectful of the unspecified transitions. In other words a minimal implementation of a KBP need not be bisimulation- or trace-equivalent to the constructed automaton, merely behaviourally-equivalent (§3.6.2).

We now demonstrate that this problem is intractable by showing that the following problem, proven by Pfleeger (1973) to be NP-complete, is equivalent to ours:

> Given an incompletely specified DFA $M = (K, \Sigma, \delta, q_0, F)$ and $k > 0$. (K and Σ are finite sets of "states" and "inputs," respectively; δ, called the "transition function," is a mapping from a subset of $K \times \Sigma$ into K; the "initial state" q_0 is in K; and the set of "final states" F is a subset of K.) Is there a way to assign a state to each unspecified transition so that the resulting complete automaton has at most k equivalence classes of states?

We can construct a protocol (see §3.6.2) from a DFA $(K, \Sigma, \delta, q_0, F)$ by defining $\mathsf{pInit}\ o = \delta(q_0, o)$ for each $o \in \Sigma$ where $\delta(q_0, o)$ is defined, and $\mathsf{pTrans}\ o\ s = \delta(o, s)$ for all $s \in K$ and $o \in \Sigma$ where $\delta(s, o)$ is defined. Let $\mathsf{pAct}\ s$ yield true iff $s \in F$. From a minimal behaviourally-equivalent protocol we can recover a minimal DFA by pruning all states from which we cannot reach a final state, i.e. one where $\mathsf{pAct}\ s$ is non-empty, in linear time. Such a DFA will contain no equivalent states, and so a minimal k is the number of its states.

Conversely a protocol A with Boolean actions determines a DFA as follows. The state space K contains a fresh state q_0 and the set of states in A, and Σ is the set of possible observations together with a fresh label l not amongst these observations. We define $\delta(s, l) = s$ for all states $s \in K$ where $\mathsf{pAct}\ A$ yields true, $\delta(s, o) = \mathsf{pTrans}\ o\ s$ for all $s \in K$ and possible observations o, and similarly $\delta(q_0, o) = \mathsf{pInit}\ o$. We set $F' = K$. (Identifying satisfaction of $\mathsf{kTest}\ f$ with finality may not be correct when there are states s where $\mathsf{kTest}\ f$ is false and there is no path from s to a final state; all states that cannot reach a final state can be discarded in a minimal automaton.)

This problem has been of interest to the electronic design automation (EDA) industry as the automata produced by high-level synthesis are often sub-optimal. Rho, Hachtel, Somenzi, and Jacoby (1994) provide an algorithm for exact solutions, and we use their `STAMINA` tool from the Berkeley Sis toolkit. We have found that it converges more quickly if we reduce the automata under bisimulation first. As this tool has proven adequate for our examples, and this issue is peripheral to the main thread of this work, we do not further discuss the mechanics of `STAMINA` or more recent work in that area. We note that we might also like to minimise the number of transitions, which is beyond the scope of `STAMINA`.

Ideally we might combine minimisation with traversal. We discuss this in §7.2.

```
robotA = agent "Robot" (kTest ("Robot" ‘knows‘ probe "inGoal"))

environment = proc halt →
  do rec pos ← (| delayAC (fromIntegerA 0 −≺ ())
                  (| muxAC (returnA −≺ halt)
                           (returnA −≺ pos)
                           ((returnA −≺ pos)
                             ‘nondetFairAC‘ (incA −≺ pos)) |) |)
     sensor ← (decA −≺ pos)
                ‘nondetAC‘ (returnA −≺ pos)
                ‘nondetAC‘ (incA −≺ pos)
     returnA −≺ (pos, sensor)

robotTop = proc () →
  do rec halted ← robotA −≺ sensor
         (pos, sensor) ← environment −≺ halted
     inGoal ← ((fromIntegerA 2 −≺ ()) ‘leAC‘ (returnA −≺ pos))
                ∧ ((returnA −≺ pos) ‘leAC‘ (fromIntegerA 4 −≺ ()))
     probeA "inGoal" −≺ inGoal
     natA (undefined :: Three) −≺ pos
     returnA −≺ (halted, pos, sensor)
```

Figure 6.2: The ADHOC model of the Robot of §2.

6.3 The Robot redux

Our first example is the familiar autonomous robot we introduced in §2. We have already constructed an implementation in §3.8.1 with an explicit-state technique; here we render it as an Arrow and find an implementation using the techniques detailed earlier in this chapter. As neither the environment nor the robot engage in any complex sequential behaviour, we describe them directly as the ADHOC circuits shown in Figure 6.2.

The Robot's KBP is quite succinct as we can represent its two actions by the Boolean result of kTest. The top-level Arrow robotTop routes the sensor reading to the agent construct.

The environment encodes the Robot's initial position as 0. At succeeding instants the position can be (fairly) non-deterministically increased by 1 provided the Robot does not perform the *halt* action, and otherwise the robot remains where it is. Independently of this a new sensor reading is generated that is within 1 of the actual position of the robot. Making the environment's decision about moving the robot fair does not affect the construction algorithm, which ignores infinitary behaviour; it simply allows us to show that the robot always halts in the goal region.

The top-level robotTop composes these two processes using the non-instantaneous recursion provided by the ArrowLoop class. This is well founded due to the use of delayAC in the environment. It also defines the `inGoal` probe, and fixes the arithmetic to be three bits wide, sufficient to represent the numbers from 0 to 7.

The automata generated by our Haskell implementation using the synchronous perfect-recall

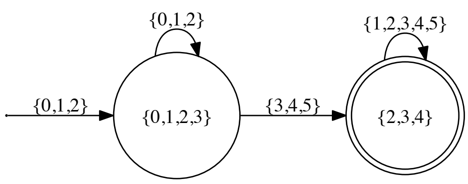


Figure 6.3: The SPR implementation of the Robot, minimised using `STAMINA`.

semantics for knowledge coincide with those from the Isabelle/HOL derivation shown in §3.8.1. Additional reduction of the synchronous perfect-recall implementation using `STAMINA` (§6.2.4) yields the automaton shown in Figure 6.3. This is what a human would design: it is a representation of the predicate *sensor* $\geq$ 3 on the domain of possible sensor readings.

We can use standard temporal model checking to verify that this implementation is adequate in the given environment. Specifically we can check that it always halts, and when it does it is in the goal region.

6.4 Logic puzzles

Knowledge-based programs can be used to solve puzzles that have an epistemic flavour; here we revisit the classic Muddy Children puzzle and show how we can find a solution to the sum and product puzzle using a custom EDSL.

6.4.1 The Muddy Children

As a further familiarisation example we render the multi-agent Muddy Children puzzle described in §3.8.2 using the broadcast combinator of §6.2.3. We need to to supply three things for each child agent: a name, an Arrow yielding an initial observation, and another Arrow for the recurring behaviour. The initial observation for child i is the announcement by the parental figure of the dirtiness of the other children paired with the presence or absence of mud on the other children's foreheads; the recurring behaviour is simply a knowledge test as for the Robot (§6.3):

```
childName, dirtyP :: Integer → AgentID
childName i = "child" ++ show i
dirtyP i = "child" ++ show i ++ "_is_dirty"

childAs = mkSizedListf (λi → (childName i, childInitObs i, childA i))
  where
    childInitObs i = second (mapSLn (λj → if i == j then zeroA else id))
    childA i = kTest (childName i `knows` dirtyP i)
```

The generator childInitObs squashes child i's observation of herself and retains the others; the zeroA combinator yields an arbitrary but fixed constant. The SizedList operation mkSizedListf:

```
mkSizedListf :: Card size ⇒ (Integer → α) → SizedList size α
```

constructs a SizedList of a width specified by its context, sourcing the elements from the provided function. The mapSLn function is the familiar map function lifted to SizedLists:

```
mapSLn :: (Arrow (⇝), Card size)
       ⇒ (Integer → α ⇝ β) → SizedList size α ⇝ SizedList size β
```

The Arrow we map across the SizedList is also handed its index, which is necessitated by the staging introduced by Arrows.

The environment incorporates the parental figure, and determines the number of children through the NumChildren type:

```
environment = proc () →
  do d ← nondetLatchAC trueA −≺ ()
     mapSLn (probeA ∘ dirtyP) −≺ d
     anyDirty ← disjoinSL −≺ d
     rec acts ← (| delayAC (replicateSL ⋘ falseA −≺ ())
                     (| (broadcast childAs)
                          (returnA −≺ (anyDirty, d ‘asLengthOf‘ acts))
                          (returnA −≺ acts) |) |)
     idSL (undefined :: NumChildren) −≺ d
     probeA "all children say yes" ⋘ conjoinSL −≺ acts
```

An arbitrary choice of muddiness is made for each child by nondetLatchAC in the first instant, after which this choice is preserved for the entire run of the system. We define dirtyP probes in the obvious way, and the proposition *anyDirty* represents the truth of the parental figure's assertion that "at least one of you has mud on your forehead." The broadcast combinator pipes the data to the childAs generator, where the common observation consists of the actions of the children in the previous instant, taken to be false before the parental figure says anything. We determine the number of children using the idSL combinator, which takes an argument with a *size* type and constrains the SizedList that passes through it.

The result of the algorithm using the SPR semantics and minimisation using `STAMINA` for three children is shown in Figure 6.4. As in §3.8.2, the initial transitions are labelled with the initial observation, i.e., the cleanliness "C" or muddiness "M" of the other two children. The dashed initial transition covers the case where everyone is clean; in the others the mother has announced that someone is dirty. Later transitions record the knowledge state of each child. Double-circled states are those in which `child1` knows her state of muddiness.

We note that the counting structure is even more pronounced here as `STAMINA` has reduced the model to a single accepting state. It has also introduced some spurious transitions in its goal

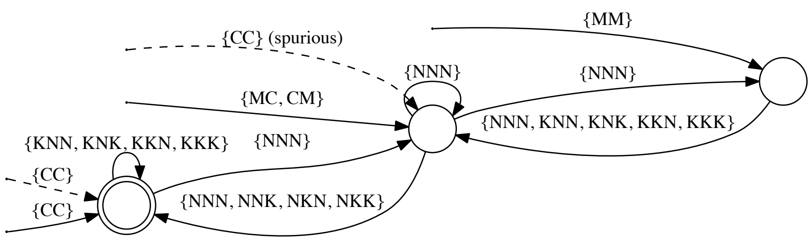


Figure 6.4: The SPR implementation of the first of three muddy children, minimised with `STAMINA`.

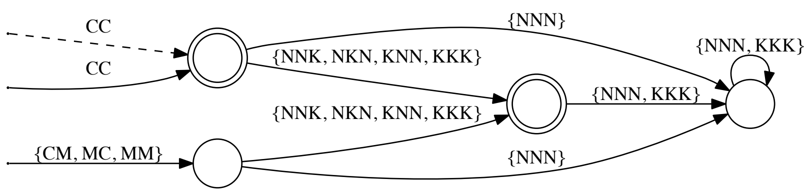


Figure 6.5: The clock automaton for the first of three muddy children, minimised using `STAMINA`.

to minimise the number of states; for instance the transition "NNN" from the accept state is impossible as all the children output "K" in that state. Moreover the initial transition marked "spurious" covers the case when everyone is clean, is told so and still don't know it.

The result of the algorithm using the clock semantics is shown in Figure 6.5. Unlike the SPR automaton this is not an adequate implementation. Intuitively the clock semantics does not record enough of the history that a full solution relies on; in this implementation a child only learns of her muddiness if it is manifestly obvious, and forgets it within two instants. The clock semantics yields essentially this automaton independently of the number of children.

Moses, Dolev, and Halpern (1986) canvas many variants of this puzzle.

6.4.2 Mr. S and Mr. P

Another venerable epistemic logic puzzle is the case of Mr S. and Mr P., also known as the sum and product puzzle. According to McCarthy (1987):

> Two numbers m and n are chosen such that $2 \le m \le n \le 99$. **Mr. S** is told their sum and **Mr. P** is told their product. The following dialogue ensues:
>
> **Mr. P:** I don't know the numbers.
>
> **Mr. S:** I knew you didn't know. I don't know either.

Mr. P: Now I know the numbers.

Mr. S: Now I know them too.

In view of the above dialogue, what are the numbers?

We seek to render this dialogue in the most direct way possible, avoiding in particular any kind of manual reasoning about what the epistemic assertions imply; that is the job of the tool.

The second step of the dialogue is an assertion about what Mr. S knew about Mr. P's state of knowledge in the previous instant, i.e., in the initial state of the dialogue. We cannot encode this directly using the syntax of §6.1, and so we introduce a new language which adds *previous state* and *present state* modalities to our knowledge formulas of type KF. As we will show, we can expand such formulas locally to an agent using delayAC (§5.2.3).

The syntax is:

```
data KFP = KFPfalse | KFPtrue |KFP 'KFPand' KFP | KFPneg KFP
         | KFPpre KF          — in the previous instant, φ.
         | KFPnow KF          — in this instant, φ.
```

We will use pre and now as synonyms for KFPpre and KFPnow respectively.

We model a dialogue as a SizedList of pairs, where a point of time is identified with a position in the list, with only one agent speaking at a time:

```
data Announcement = AgentID :> KFP
infix 0 :>
type Dialogue w = SizedList w Announcement
```

The SizedList is forced on us by the need to propagate size constraints using types, which we discuss further below. Intuitively we evaluate the announcement for the current time step and broadcast its truth value, with the passive agent "nodding along" with the active one by making the always-true statement. Our specific dialogue, using the abbreviation knows_a^{mn} for $\widehat{\mathsf{knows}}_a\ m \land \widehat{\mathsf{knows}}_a\ n$, is rendered as follows:

```
dialogue :: Dialogue Four
dialogue = mkSizedListA [
    — Mr. P: I don't know the numbers.
  mrP :> now (notAC knows^mn_mrP),
    — Mr. S: I knew you didn't know. I don't know either.
  mrS :> pre (knows_mrS (notAC knows^mn_mrP)) ∧ now (notAC (knows^mn_mrS)),
    — Mr. P: Now I know the numbers.
  mrP :> now (knows^mn_mrP),
    — Mr. S: Now I know them too.
  mrS :> now (knows^mn_mrS) ]
```

As we expect participants in a dialogue to have perfect recall, we translate such a Dialogue into a SizedList of Arrows that can be passed to the broadcast combinator (§6.2.3). The scheme is shown in Figure 6.6.

The functions kfpToKF kt compile a KFP formula into a KF formula and an Arrow handles the temporal dimension. We define it using the standard state Monad StateM (§5.1.2) so we can common-up the knowledge subformulas in the dialogue. (This reduces the solution of the Mr. S and Mr. P dialogue to the construction of three automata.)

The dialogueForAgent function assembles an Arrow for agent *aid* based on his part of the dialogue. We add the state of the dialogue (a counter) to the common observation. We omit the class context as it is large and unedifying. The *encS* function encodes a single line of the dialogue: if it is the active one, and the present agent is speaking, then it returns the value of the relevant kTest as constructed by *kTests*, otherwise it returns trueA, i.e. that *aid* has (successfully) said nothing. As a dialogue is finite, we take all statements after it completes to be true.

The runDialogue combinator combines these Arrows with a SizedList of pairs of agent identifiers and arrows giving their initial observation, and composes these with the broadcast combinator of §6.2.3:

```
runDialogue :: ...
            ⇒ SizedList size (AgentID, env ⇝ iobs) → Dialogue dlen
            → (env ⇝ SizedList size B(⇝))
runDialogue agents (d :: Dialogue dlen) = proc ienv →
  do rec dState ← (| delayAC (constNatA (undefined :: dlen) 0 −≺ ())
                         ( (constNatA (undefined :: dlen) (dlen − 1) −≺ ())
                            ‘minAC‘ (incA −≺ dState) ) |)
     rec acts ← (| delayAC (replicateSL ⋘ trueA −≺ ())
                       (| (broadcast aars) (returnA −≺ ienv)
                                           (returnA −≺ (dState, acts)) |) |)
     returnA −≺ acts
  where
    dlen = c2num (undefined :: dlen)
    aars = mapSL (λ(aid, iarr) → (aid, iarr, dialogueForAgent aid d)) agents
```

The variable *dState* tracks our progress through the dialogue; it is much wider than it needs to be as we cannot readily take logarithms using our sizes-in-types technology (see §5.3). Note also that we consider the empty dialogue to be successful.

The actions of the agents (the truth of their statements) are broadcast to all agents, in the same way as for the Muddy Children (§6.4.1).

Returning to our specific scenario, the agents initially observe the product or sum of the pair of numbers, as appropriate:

```
agentInitAs = mkSizedListA [ (mrS, numCastA ⋘ addA), (mrP, mulA) ]
                           ‘withLength‘ (undefined :: Two)
```

```
kfpToKF :: (ArrowComb (⇝), ArrowDelay (⇝) (B(⇝)))
        ⇒ KFP → StateM [KF] ([B(⇝)] ⇝ B(⇝))
kfpToKF = go
  where
    go f = case f of
             KFPfalse → return falseA
             KFPtrue → return trueA
             x `KFPand` y → liftM2 andAC (go x) (go y)
             KFPneg x → liftM notAC (go x)
             KFPnow (KFneg x) → liftM notAC (go (now x))
             KFPnow x → kt x
             KFPpre (KFneg x) → liftM notAC (go (pre x))
             KFPpre x → liftM (delayAC falseA) (kt x)


kt :: Arrow (⇝) ⇒ KF → StateM [KF] ([B(⇝)] ⇝ B(⇝))
kt f0 = StateM (findKT 0)
  where
    findKT :: Arrow (⇝) ⇒ Int → [KF] → ([B(⇝)] ⇝ B(⇝), [KF])
    findKT n [] = (arr (λxs → xs !! n), [f0])
    findKT n ffs@(f : fs)
           | f0 == f = (arr (λxs → xs !! n), ffs)
           | otherwise = second (f :) (findKT (n + 1) fs)


dialogueForAgent :: ...
                 ⇒ AgentID → Dialogue dlen
                 → ((iobs, (Nat dlen (⇝), cobs)) ⇝ B(⇝))
dialogueForAgent aid (d :: Dialogue dlen) = proc (iobs, (dState, cobs)) →
   do kts ← kTests −≺ ()
      dA −≺ (dState, kts)
  where
    (dA, kts) = runStateM encD []
    encD = foldM encS trueA (zip [ 0 .. ] (unSizedListA d))
    encS restA (i, aid′ :> statement) =
      do sA ← if aid == aid′ then kfpToKF statement else return trueA
         return $ proc (s, kts) →
           (| muxAC ( (returnA −≺ s) `eqAC` (constNatA (undefined :: dlen) i −≺ ()) )
                    (sA −≺ kts)
                    (restA −≺ (s, kts)) |)
    kTests = foldr (liftA2 (:) ∘ kTest) (arr (const [])) kts
```

Figure 6.6: Functions for converting a dialogue into an agent protocols that can be passed to the broadcast combinator.

The sum is left-padded with zeroes to make it the same width as the product, as initial observations must have a uniform type.

The top-level simply chooses two numbers non-deterministically subject to the constraints of the puzzle, attaches two probes and runs the dialogue:

```
top = proc () →
  do mn ← (| nondetLatchAC (λs0 → initA −≺ s0) |)
     probeA mvar *** probeA nvar −≺ mn
     runDialogue agentInitAs dialogue −≺ mn
 where
   natAW = natA (undefined :: Seven)
   initA = proc (m, n) →
           ((fromIntegerA 2 −≺ ()) ‘leAC‘ (natAW −≺ m))
             ∧ ( (returnA −≺ m) ‘leAC‘ (returnA −≺ n) )
             ∧ ( (returnA −≺ n) ‘leAC‘ (fromIntegerA 99 −≺ ())) |)
```

We need Seven bits to represent the hundred numbers of interest. The two probes define the variables m and n mentioned in the dialogue.

Unfortunately this model does not converge within reasonable time. One might suspect that this is due to the multiplication causing the BDDs to explode (Bryant 1991), but modern hardware can readily construct BDDs for multipliers of seven bits and more. The explosion is closely related, however: constructing the BDD that represents the quotient of the initial states, i.e. the BDD that represents $x * y = x' * y'$, takes a very long time.

We ameliorate this issue by explicitly computing the sums and products using Haskell's much faster machine arithmetic and including them in the state:

```
top = proc () →
  do (mn, sp) ← (| nondetLatchAC (λs0 → disjoinAC initAs −≺ s0) |)
     probeA mvar *** probeA nvar −≺ mn
     runDialogue agentInitAs dialogue −≺ sp
 where
   natAW = natA (undefined :: Seven)
   natAWM = natA (undefined :: Fourteen)
   initAs = [ proc ((m, n), (s, p)) →
                ( ((natAW −≺ m) ‘eqAC‘ (fromIntegerA vm −≺ ()))
                ∧ ((natAW −≺ n) ‘eqAC‘ (fromIntegerA vn −≺ ()))
                ∧ ((natAWM −≺ s) ‘eqAC‘ (fromIntegerA (vm + vn) −≺ ()))
                ∧ ((natAWM −≺ p) ‘eqAC‘ (fromIntegerA (vm ∗ vn) −≺ ())) )
                | vm ← [2 .. 99], vn ← [vm .. 99] ]
```

The Arrow combinator disjoinAC folds orA across a list of Arrows, which in this case are constraints on the initial state s_0.

This reduces both agents' initial observations to the appropriate projections of the derived components:

```
agentInitAs = mkSizedListA [ (mrS, arr fst) , (mrP, arr snd) ]
                           `withLength` (undefined :: Two)
```

The rest of the description remains as it was.

A solution to this puzzle is the initial state of a trace t on which all the statements are true, i.e. when both Mr. S and Mr. P always say yes; call this proposition p. We therefore seek a witness (a run) for the LTL formula $\Box p$. As our model checker uses CTL (§2.3), we seek a counter example to the claim that on all paths there is always a time when $\neg p$, i.e. to the formula $\mathsf{AF}\ \neg p$.

On a modern machine (a 2011 MacBook Pro with an Intel Core i7 2.2GHz processor), when compiled with the Glasgow Haskell Compiler (GHC) 7.4.1 this model converges in about four minutes and yields the expected solution. Most of the time is spent in the automata construction, and in particular in the BDD operations, which are memory-bandwidth intensive and hence very sensitive to memory contention. There is certainly scope for more optimisation.

6.4.3 Concluding remarks

van Ditmarsch et al. (2008) provide some history of this puzzle and show how DEMO (§2.1.1) can find a solution and prove its uniqueness. The model written by Ji Ruan does this in approximately six and a half minutes when compiled with GHC 7.4.1 on the same machine as above. When run in the Haskell interpreter GHCi, this code does not converge in two hours.

Luo et al. (2008) verify that the solution to this puzzle is valid and unique with their MCTK model checker. This presupposes that the answer to the puzzle is known.

The original formal treatment of this puzzle was by McCarthy (1987), who used ad hoc modalities encoded into first-order logic. This is clearly not an algorithmic approach in general. A solution using ad hoc number-theoretic reasoning, backtracking and memoisation is given by Kiselyov (2006), who claims that his approach is more concise than McCarthy's. This list-Monad based solution in Haskell can find the solution and demonstrate its uniqueness in less than half a second.

van Ditmarsch et al. (2008, §9) declared that "an extension [of MCK] is called for", and ADHOC could be seen as that extension. We contend that our modelling is the most natural of any of this puzzle, though we grant that the automata-theoretic algorithm used here is not well-suited to data-centric problems like this one, and it is unfortunate that BDDs have some pathologies on multipliers and other circuits that are difficult to overcome in general. However we can take some comfort from having the full power of Haskell available to ameliorate these issues when they arise. This is a major benefit of the EDSL approach.

It is not difficult to add fully general past-time temporal operators to the language of the guards, using the standard technique of augmenting the state with extra propositions for temporal subformulas. This would require modifying ADHOC itself as the propositions need not be local to an agent. We will not pursue this further as this is the only example we will present that needs this kind of modality.

Smullyan (1982) provides many more epistemic puzzles, including what he calls "meta-puzzles": the fact that the puzzle was solved by someone else is key to being able to solve the puzzle. Most puzzles like Smullyan's only require us to reason about an agent's knowledge in isolation, whereas the Mr. S and Mr. P puzzle depends on reasoning about what is mutually known, at least as we have modelled it.

The reader may care to attempt this puzzle:

> Two old friends, Ernie and Bernie, bump into each other in the street. It was more than twenty years since their last meeting, so they decide to spend some time together in a nearby bar, chatting about their lives. At some point, Bernie asks the inevitable: "So, you got married?"
>
> "Well, yes!" — enthusiastically claims Ernie. "And I have three beautiful children!"
>
> "That's great, Ernie! And how old are your kids?" — enquired Bernie.
>
> "I know you are a sucker for puzzles, Bernie, so I will let you figure it out through a few clues. You should not have any trouble getting it. First clue: If you add the ages of my kids, the result is thirteen."
>
> "Whoa! That doesn't help me much, does it?" — complains Bernie. "Could you give me another clue?"
>
> "Sure, a second clue: If you multiply their ages together, the result is the same as how much we payed for these beers."
>
> Bernie scratches his head for a minute, and cannot figure it out yet... Before he complains again, Ernie realizes the mistake, apologizes, and offers Bernie the last clue: "My oldest plays the piano!"
>
> Bernie had no trouble finding the ages of the children this time.

We conclude by noting that these sorts of logic puzzles typically involve reasoning about the epistemic states of the agents in an unchanging situation, which is a simpler problem than the KBP formalism is intended to treat; in general the state in a KBP scenario can non-deterministically change at each point in time.

6.5 Model checking the Dining Cryptographers

No discussion of a tool for epistemic reasoning can avoid a treatment of the Dining Cryptographers. Chaum (1988) sets the scene:

> Three cryptographers are sitting down to dinner at their favorite three-star restaurant. Their waiter informs them that arrangements have been made with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S. National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if NSA is paying. They resolve their uncertainty fairly by carrying out the following protocol:
>
>> Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer on his right, so that only the two of them can see the outcome. Each cryptographer then states aloud whether the two coins he can see — the one he flipped and the one his left-hand neighbor flipped — fell on the same side or on different sides. If one of the cryptographers is the payer, he states the opposite of what he sees. An odd number of differences uttered at the table indicates that a cryptographer is paying; an even number indicates that NSA is paying (assuming that the dinner was paid for only once). Yet if a cryptographer is paying, neither of the other two learns anything from the utterances about which cryptographer it is.

As Chaum goes on to prove, this protocol is *information-theoretically secure*: no matter how computationally powerful the agents are, the anonymity of a cryptographer who paid is assured, provided the others stick to the protocol. This is not generally true of cryptographic schemes where secrecy depends on computational complexity.

This protocol is a staple of epistemic model checkers (Kacprzak, Lomuscio, Niewiadomski, Penczek, Raimondi, and Szreter 2006; Lomuscio et al. 2009; Su et al. 2007; van der Meyden and Su 2004). Here we build a very succinct model using ADHOC, shown in Figure 6.7, which leads to a very efficient verification of the protocol. We encode who paid with a number represented as a binary word of width ArithmeticWidth. The agents are numbered from 1 to NumAgents, with the NSA being notionally agent 0. The key to our brevity is recognising that the intermediate local steps in the protocol are not relevant from the perspective of information flow between the agents; in a typical model of this protocol all the agents do not perform any externally-visible actions during these steps, of which there are a fixed number.

Who paid is determined by the nondetChooseAC combinator (§5.2.5):

$$\begin{aligned}\mathsf{nondetChooseAC} :: {} & (\mathsf{ArrowLoop}\ (\rightsquigarrow),\ \mathsf{ArrowNonDetInst}\ (\rightsquigarrow)\ \upsilon) \\ & \Rightarrow ((\gamma,\ \upsilon) \rightsquigarrow \mathsf{B}(\rightsquigarrow)) \to (\gamma \rightsquigarrow \upsilon)\end{aligned}$$

This combinator non-deterministically chooses an object of type υ that satisfies the predicate at every instant. The coin flips are made by the nondetBitA combinator. All choices are recorded in the state.

We also use the fanoutSLn combinator:

$$\mathsf{fanoutSLn} :: (\mathsf{Arrow}\ (\rightsquigarrow), \mathsf{Card}\ \mathit{size}) \\ \Rightarrow (\mathsf{Integer} \to (\gamma, \alpha) \rightsquigarrow (\alpha, \beta)) \to (\gamma, \alpha) \rightsquigarrow (\alpha, \mathsf{SizedList}\ \mathit{size}\ \beta)$$

which is similar to the row combinator (§4.2.1) in that the values of type a (in this case the coins) are fed from left to right. We use the non-instantaneous loop combinator (which underpins the **rec** syntax) to feed the rightmost coin to the leftmost cryptographer; this is sound as the coin's value is recorded in the state. Moreover the final broadcast represented by the variable *said* is also implemented using this loop. This is well founded as the agents merely observe it.

The netlist for three cryptographers is shown in Figure 6.8.

A state in this model is described by $n + \lg n$ bits for n cryptographers: one bit per agent to represent the coin flips and enough bits to record who paid. As we are in a temporal setting, we also need a transition relation which is represented in the standard manner of having present and next state variables for each bit in the state. Therefore the model uses essentially $2n + 2\lg n$ BDD variables in total.

This approach reduces the verification problem to checking an instantaneous function of the state; as there are no intermediate states between the coin flips and the broadcast, it is sound to use the observational semantics for knowledge directly (see §2.3.2). Luo, Su, Gu, Wu, and Yang (2010) also use the observational semantics, but as observed by Al-Bataineh and van der Meyden (2011, §12), the extra steps in their protocol require history variables to be manually added, and these need to be separately shown to be adequate for the anonymity claim to hold.

Our specifications are as follows, using the probes defined in Figure 6.7, and taking the perspective of the first cryptographer. The CTL operator AG means we ask that the specification be invariant on all runs.

Firstly, if the cryptographer paid then she knows the entire paid vector:

$$\mathsf{spec}_1 \equiv \mathsf{AG}\ (\mathtt{paid}_{dc1} \longrightarrow \widehat{\mathsf{knows}}_{dc1}\mathtt{paid})$$

Secondly, if the cryptographer didn't pay, then she doesn't know who paid. This is false.

$$\mathsf{spec}_2 \equiv \mathsf{AG}\ (\neg\mathtt{paid}_{dc1} \longrightarrow \neg\widehat{\mathsf{knows}}_{dc1}\mathtt{paid})$$

The final correctness assertion says that if this cryptographer didn't pay then she ultimately knows either that the NSA paid, or that one of the other cryptographers did but not which one.

$$\mathsf{spec}_3 \equiv \mathsf{AG}\ (\neg\mathtt{paid}_{dc1} \longrightarrow \quad \mathsf{knows}_{dc_1}\mathtt{The\ NSA\ paid} \\ \vee\ (\mathsf{knows}_{dc_1}\neg\mathtt{The\ NSA\ paid} \wedge \neg\widehat{\mathsf{knows}}_{dc_1}\mathtt{paid}))$$

We could also use the bounded perfect recall semantics of MCK (§2.3.2) but as the state consists entirely of non-deterministic choices that are made afresh every instant, perfect recall and the instantaneous observational semantics for knowledge coincide.

```
type NumAgents = Three
type ArithmeticWidth = Two

dcAgentName :: Integer → AgentID
dcAgentName i = "dc" ++ show i

dcPaid :: Integer → ProbeID
dcPaid i = "paid" ++ show i

  — The agent observes whether or not she herself paid, the coin flip
  — to her left and the broadcast.
dcAgent i = proc ((said, whoPaid), lCoin) →
 do ia ← fromIntegerA i −≺ ()
    iPaid ← probeA (dcPaid i) ⋘ eqA −≺ (whoPaid, ia)
    agent (dcAgentName i) dcA −≺ (iPaid, lCoin, said)
  where
    dcA = proc (iPaid, lCoin, said) →
     do rCoin ← nondetBitA −≺ ()
        say ← xorA ⋘ second iffA −≺ (iPaid, (lCoin, rCoin))
        returnA −≺ (rCoin, say)

  — The environment.
environment = proc () →
 do paid ← (| nondetChooseAC (λv →
       do numAgents ← constNatA (undefined :: ArithmeticWidth) n −≺ ()
          leA −≺ (v, numAgents)) |)
    probeA "paid" −≺ paid
    probeA "The NSA paid" ⋘ eqA ⋘ fromIntegerA 0 &&& id −≺ paid
      — The dining cryptographers sit in a circle.
    rec (coin, said)
          ← fanoutSLn dcAgent
            −≺ ((said ‘withLength‘ (undefined :: NumAgents), paid), coin)
  where
    n = c2num (undefined :: NumAgents) :: Integer
```

Figure 6.7: The dining cryptographers as an ADHOC circuit.

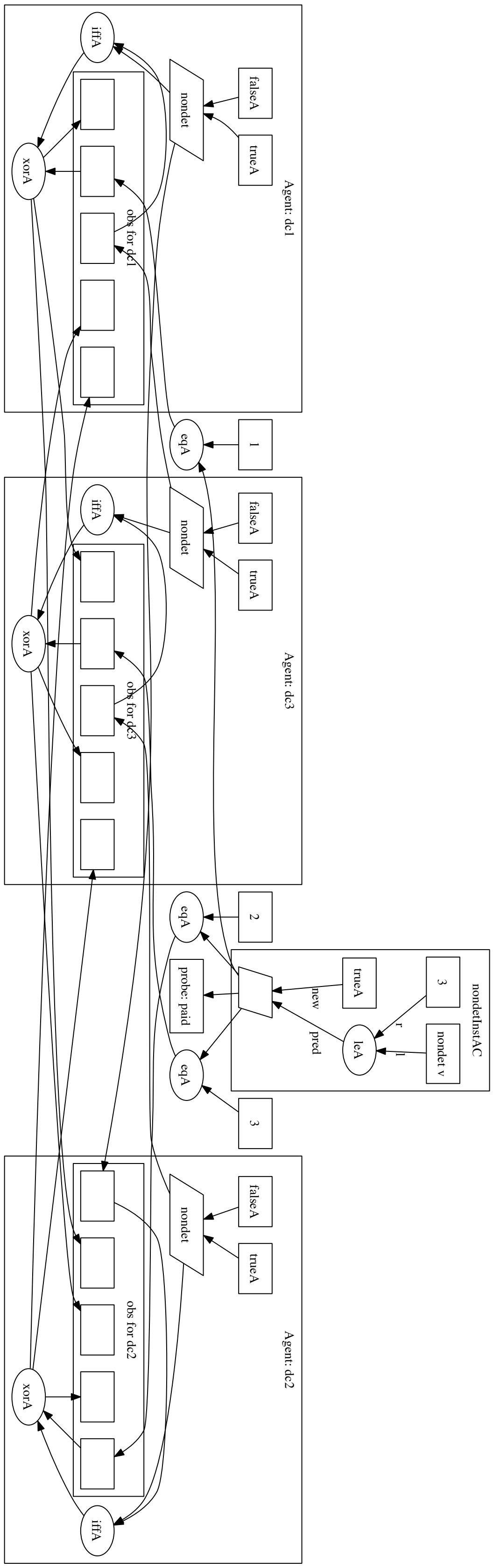


Figure 6.8: The circuit for three dining cryptographers.

Our model is faster to verify than any of the published ones. Using the GHC 7.4.1 Haskell compiler[1], on a 2011 MacBook Pro with an Intel Core i7 2.2GHz we can check all three specifications for 80 dining cryptographers in 1.4 seconds using approximately 940,000 BDD nodes with variable reordering disabled. In contrast, the encoding of Su et al. (2007) uses 6 variables per cryptographer, and their MCTK tool takes just over a minute on a Pentium 1.6GHz processor using about 125,000 BDD nodes to verify the last of these specifications[2]. MCTK uses the same BDD package (CUDD due to Somenzi et al.) as we do, with some type of variable reordering heuristic enabled. This shows that there need be no correlation between the number of BDD nodes used and the speed of the verification.

The epistemic model checker MCMAS contains some sophisticated optimisations (Lomuscio et al. 2009, §4). Its authors claim it can check 18 agents in approximately 48 minutes on a 2.2GHz Core 2 Duo. We note that the reachable part of the state space in this example is a tiny part of the total, so the number of states is not much of a guide to how well the tool scales.

The review by Kacprzak et al. (2006) shows that the SAT-based checker VerICS is not competitive, struggling to verify even 6 cryptographers.

Enabling the "sift" variable reordering heuristic in CUDD makes our 80-agent verification take approximately 30 seconds using 240,000 live nodes at the peak. This reflects the hazard that benchmarking BDD-based algorithms may degenerate to timing these heuristics.

ADHOC can verify 300 dining cryptographers in about 200 seconds without variable reordering, and 400 agents in about 500 seconds. This demonstrates that our verification does not run in time linear in the number of agents.

We conclude by reiterating the observation of Su et al. (2007) that efficiently model checking the Dining Cryptographers is more about finding a good encoding of the model than the checker itself. That our Haskell model is faster than similar tools written in C and C++ shows that the computation burden is carried by the BDD package and not the code orchestrating those operations (which is as it should be). We also note that the symmetry reduction advocated by Cohen, Dam, Lomuscio, and Qu (2009) is likely to be a big win in some scenarios, but the Dining Cryptographers is not the example to show that.

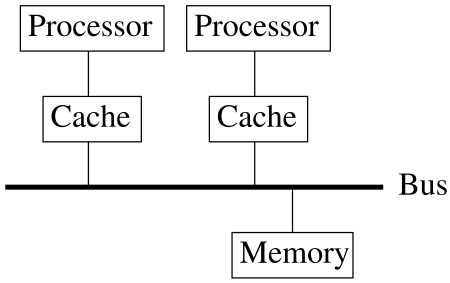
6.6 Cache coherency protocols

Our final example is a knowledge-based treatment of a class of cache coherency protocols, following Baukus and van der Meyden (2004). These protocols provide some kind of consistent view of a distributed shared-memory system to multiple processors, with the goal of ameliorating the von Neumann bottleneck by keeping data close to where it is processed.

[1] As we remarked in §6.4.3, some code under the Haskell interpreter GHCi may be orders of magnitude slower than the code produced by the compiler. The compile time is not included in our results.

[2] I attempted to rerun their benchmarks on recent hardware but could not get MCTK to build.

We focus here on hardware-based bus protocols as depicted on the right, and in particular the MOESI protocols that have been surveyed by Archibald and Baer (1986) and Handy (1998); see also Sweazey and Smith (1986). As all communication is by broadcast, the cache controllers can *snoop* on each others' bus activities. These controllers can have quite subtle epistemic properties and hence optimisation possibilities.



Baukus and van der Meyden (2004) give a knowledge-based *specification* for this class of protocols. Such specifications generalise KBPs by abstracting actions to pairs of epistemic pre- and post-conditions, and by using *local propositions* instead of a knowledge modality, implementations of the epistemic conditions need only be sound and not complete. In other words, an implementation can be simplified by acting as though it does not know something when in fact it does. They claim that all implementations of their specification provide a strongly sequential view of memory, and proceed to show that three industrial protocols do satisfy it using the asynchronous perfect-recall semantics implemented by MCK (§2.3.2). They also claim to show that these protocols make optimal use of their knowledge, i.e., that the tests corresponding to the knowledge conditionals in the specification are in fact equivalent to them, from which we can infer that bus communication is minimised.

This is an appealing target for our algorithmic techniques as the hardware setting guarantees that there are only finitely-many states, and as we can consider the bus to be a synchronous broadcast mechanism we can use the perfect recall semantics of §3.7.2. It also serves as a more realistic test of the tool as this example is control centric and we do not expect to encounter the pathological BDD behaviour we saw in §6.4.2. Our KBPs will, however, need to be significantly more concrete than their specification.

The following sections present our model and conclude with a comparison of our approach to Baukus and van der Meyden (2004) and other work in the literature.

6.6.1 Kesterel model

While there is a large body of work on formal models of cache coherency protocols, there is little on deriving them from high-level specifications as proposed by Baukus and van der Meyden (2004). Pointers into the former can be found in the early survey by Pong and Dubois (1996).

We avail ourselves of some standard abstractions following Clarke, Grumberg, Hiraishi, Jha, Long, McMillan, and Ness (1995). Concretely we model only the steady state of the cache coherency protocol and not the initialisation protocol or a cache directory that maps memory addresses to multiple cache lines. We abstract the cache line under consideration to a single bit, which is justified as the cache controllers associate state with the entire line and not the individual words. We treat just two cache controllers here although the model is parametric. As the agents' protocols are stateful we use Kesterel (§5.5).

The MOESI protocols are so-called due to the states that cached data can be in; see Sweazey and Smith (1986) for details. Here we identify the key idea of *ownership* with dirtiness: a cache owns the line precisely when the value of its locally-modified copy is unknown to the other caches. In MOESI parlance we have an *exclusive modified* (M) state but not one for *shared modified* (O). Our knowledge-based treatment implicitly handles the *invalidity* (I) and *unmodified sharing* (S) of the line. We do not treat the *exclusive unmodified* (E) state, where a clean cache knows it has the only copy of the line, for reasons we discuss further in §6.6.3.

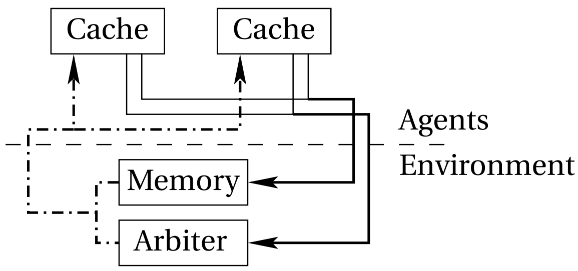


Figure 6.9: The arbitrated bus.

We treat the bus as a broadcast environment as shown in the figure on the right. As we mentioned earlier we optimise bus communication by adopting a perfect-recall semantics for knowledge. The processors non-deterministically issue read or write requests, or perform local computations, and are synchronously composed with their cache controllers. (This is not to say that the processors proceed synchronously, but that their memory interfaces do.) Finally, as we are not modelling the initialisation protocol, we assume that the cache controllers are independently initialised. With this in mind we use the construction algorithm of §6.2.3 with the streamlined representation of §3.7.5 for perfect recall in independently-initialised broadcast scenarios where agents are non-deterministic.

The following sections detail our model of the Write-Once protocol (Archibald and Baer 1986, §2.1) that omits the *reserved* (E) state as we mentioned previously. We begin by discussing the timing issues in the model, and how we model the arbitrated bus that the caches communicate over. The latter involves lifting the machinery for KBPs of §6.1 to Kesterel. We proceed to describe the KBPs for the cache controllers and the properties of the implementations we construct.

A note on timing

As our model is globally synchronous we have to play close attention to the timing behaviour of the components. We make use of the following *register* construct, whose interface is given in the HOAS-style of signalE:

$$\begin{aligned}\mathsf{registerE} :: (\mathsf{EC}\ (\rightsquigarrow),\ \mathsf{ArrowProbe}\ (\rightsquigarrow)\ (\mathsf{B}(\rightsquigarrow)))\\ \Rightarrow \mathsf{ProbeID} \to ((\mathsf{Signal}, \mathsf{Signal}, \mathsf{Signal}) \to \mathsf{E}\ (\rightsquigarrow)\ ()\ ()) \to \mathsf{E}\ (\rightsquigarrow)\ ()\ ()\end{aligned}$$

The three signals respectively set the register, reset the register and reflect the state of the register, which is instantaneously updated by the set and reset actions. If both are present, the reset action takes precedence over the set action. If both are absent then the register retains its value from the previous instant. A probe with the given identifier is attached to the logic that generates the value of the register in the present instant.

The register_delayedE construct has the same interface but defers the effects of the actions to the next instant; the probe again reflects the value of the register in the present instant, i.e. before the delayed effects occur. We provide these constructs as primitives simply to save space over the Kesterel renditions.

An arbitrated bus

Our first combinator composes the environment and the cache agents as shown in Figure 6.9. We cannot use standard Kesterel signals (§5.5.2) for communication amongst the agents as we cannot allow their behaviour to be instantaneously mutually dependent without violating the assumptions of our construction algorithm (§3.7.5). Therefore we use a non-standard semantics that is similar to that of registerE that we describe below. We take it as axiomatic that at most one agent process can be using the bus at a time.

This combinator is built out of two smaller ones. Firstly we lift the broadcast combinator of §6.2.3 to Kesterel:

$$\begin{aligned}&\mathsf{broadcastE} :: (\mathsf{ArrowBroadcast}\ (\leadsto)\ \mathit{iobs}\ (\mathsf{Cin}\ (\mathsf{B}(\leadsto)),\ \mathsf{Esigs}\ (\mathsf{B}(\leadsto))),\ \mathsf{EC}\ (\leadsto),\ \mathsf{Card}\ \mathit{size})\\ &\qquad\Rightarrow (\mathsf{SizedList}\ \mathit{size}\ (\mathsf{AgentID},\ \mathsf{E}\ (\leadsto)\ ()\ ()))\ \to\ \mathsf{E}\ (\leadsto)\ ()\ ()\end{aligned}$$

The effect of this combinator is to compose the agent processes into a single Kesterel computation, ensuring that they only communicate via Kesterel signals that are uniformly broadcast. The signature given here is a simplification of the interface to the full combinator; in the present setting the agents make no initial observations. Here we employ the StructureDest class instead of Structure as it allows us to handle the recursive type Esigs (see §5.3). The termination and exception behaviour of the agent processes is ignored as we expect these to never terminate.

Our second combinator implements the communication pattern shown in Figure 6.9:

$$\begin{aligned}&\mathsf{busE} :: (\mathsf{EC}\ (\leadsto),\ \mathsf{Structure\ Signal}\ v)\\ &\quad\Rightarrow (v \to (\mathsf{E}\ (\leadsto)\ ()\ (),\ \mathsf{E}\ (\leadsto)\ ()\ ()))\quad \text{— (agent processes, environment)}\\ &\quad\to ()\leadsto ()\end{aligned}$$

The argument function yields the agent process and the environment process. (We construct the former using broadcastE.) As with signalE, it allocates as many signals as required to fill the structure v. The semantics of these signals is non-standard, however: the environment process is given the outgoing signal environment of the agent process, and conversely the agent process receives the outgoing signal environment of the environment process delayed by an instant. This means that the environment can react to the agents' actions within the same instant, but the agents communicate with a unit delay; one can see this as a literal encoding of the interpreted systems model of §3.4.

We adopt this semantics for two reasons. Firstly, it avoids races between the memory and cache controller processes, as we discuss later. Secondly, the construction algorithm of §3.7.5 requires us to record the actions of the agents in the states and broadcast these at the next instant; this implies that we cannot allow the environment and agents to instantaneously communicate.

Finally our top-level combinator incorporates bus arbitration:

```
arbitratedBus :: (Card size, EC (⇝), Structure Signal v,
                  ArrowBroadcast (⇝) () (Cin (B(⇝)), Esigs (B(⇝))))
              ⇒ (v → (SizedList size (AgentID, (Signal, Signal) → E (⇝) () ()),
                       E (⇝) () ()))
              → () ⇝ ()
```

The agent processes are composed using broadcastE, and with the environment using busE. The two extra signals arbitratedBus passes to each agent are used to request and receive access from the bus arbitration logic, which we discuss further in the next section. Note that these signals are necessarily broadcast, i.e. all agents know which of them is requesting and has access to the bus. This is somewhat natural when the cache controllers are bus peers, but less so when the bus is a daisy chain of devices.

The environment processes

The environment process consists of the bus arbiter and the central memory. The arbiter is an adaptation of the classic de Simone cyclic design (Potop-Butucaru et al. 2007, §2.3), which has the traditional mutual exclusion properties such as the absence of unnecessary waiting.

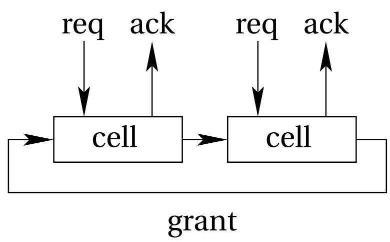


Figure 6.10: The arbiter.

The architecture is shown in Figure 6.10. Our variant rendered in Kesterel is shown in Figure 6.11. We use a few custom constructs in addition to the standard Esterel ones; in particular we adapt the Haskell Monadic combinators unless and when to the Kesterel setting, where the null action is nothingE. The form loopE_ f abbreviates catchE (loopE ∘ f), i.e. it allocates an exception that can be used to exit the loop. The function rotateSL shifts the head of a SizedList to its end. We use the idiom whenE′ $c\,f$ to test for the Boolean condition c in the underlying Arrow (⇝); the use in cell allows us to exit the inner loop if either signal is present. The standard Haskell idiom $f\ \$\ \lambda x \rightarrow g$ allows us to avoid parenthesises when allocating signals and exceptions in the scope g.

In contrast to the de Simone arbiter, our design grants access to a process for as long as its request signal is active. This makes the unconditional weak fairness of the original design dependent on the agents relinquishing the bus.

The arbiter cells circulate a token, which is recorded here in the control state of the cells. When a cell possesses the token it checks if its associated process has emitted $request_{in}$, i.e., requested

```
cell :: (EC (⇝), ArrowProbe (⇝) (B(⇝)))
    ⇒ Signal → ((Signal, Signal), (Signal, Signal)) → E (⇝) () ()
cell active ((grant_in, grant_out), (request_in, ack_out)) = loopE body
  where
    body = catchE $ λexn →
      whenE grant_in
        (presentE request_in
          ((loopE_ (λexn′ → emitE ack_out ⋙ emitE active ⋙ pauseE
                              ⋙ unlessE request_in (throwE exn′))
           ⋙ (loopE (emitE grant_out
                        ⋙ whenE′ (sigE request_in ∨ sigE active) (throwE exn)
                        ⋙ pauseE))))
          (emitE grant_out))
       ⋙ pauseE


cells :: (Card size, EC (⇝), ArrowProbe (⇝) (B(⇝)))
    ⇒ SizedList size (Signal, Signal) → E (⇝) () ()
cells reqAckSL = signalE $ λ(gtSL, active) →
  let csSL = zipWithSL (cell active)
                       (zipWithSL (arr id) (gtSL, rotateSL gtSL), reqAckSL)
      g0 : _ = unSizedListA gtSL
      initCell = loopE_ (λexn → emitE g0 ⋙ whenE active (throwE exn) ⋙ pauseE)
  in foldr (||||) initCell (unSizedListA csSL)
```

Figure 6.11: The Kesterel code for the arbiter.

the bus. If not it forwards the token to the next cell by emitting its $grant_{out}$ signal. Otherwise it grants the bus to the process by emitting ack_{out} until the instant when the process ceases to emit $request_{in}$. The cell then grants the token to the next cell until it receives a new request (which it can process instantaneously) or some other cell claims the token by emitting *active*.

Unfortunately the basic circuit translation (§5.5.2) mistranslates this Kesterel program as it is schizophrenic. In particular, the exception *exn* in body is instantaneously reincarnated after this series of events: a process gains access to the bus, relinquishes the bus, and is granted access again without another process gaining access in between. We resolve this by duplicating the loop body, i.e., defining cell to be loopE (body ⋙ body).

The other component in the environment is the memory process shown in Figure 6.12 with the Kesterel implementation shown in Figure 6.13. The sigCaseE construct executes the second component of the first alternative *sig* :− *f* in its argument list whose *sig* is present, and executes nothingE when all are absent.

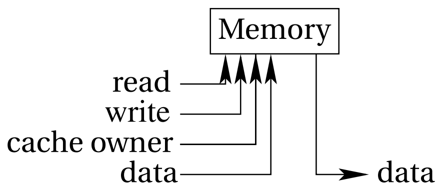


Figure 6.12: The memory process.

This process responds to requests for the memory value, but only when no cache claims own-

```
memory :: (EC (⇝), ArrowProbe (⇝) (B(⇝)))
          ⇒ (Signal, Signal, Signal, Signal) → E (⇝) () ()
memory (cache_owner, mem_read, mem_write, mem_val) =
    register_delayedE "memory value" $ λ(reg_set, reg_reset, reg_val) →
    loopE $ sigCaseE
      [ mem_read : – pauseE ⋙ presentE cache_owner
            — A cache has claimed ownership, record what it says
          (presentE mem_val (emitE reg_set) (emitE reg_reset))
            — Caches silent, emit our value
          (whenE reg_val (emitE mem_val))
      , mem_write : – presentE mem_val (emitE reg_set) (emitE reg_reset)
      ] ⋙ pauseE
```

Figure 6.13: The Kesterel code for the memory process.

ership, which requires it to wait for an instant. By exploiting the timing semantics of the busE combinator the system can process read requests in two cycles whoever owns the cache line. The register is updated on write requests and read requests where a cache claims ownership.

The cache controller KBPs

In this scenario an agent is the composition of a processor and a cache controller as shown in Figure 6.14.

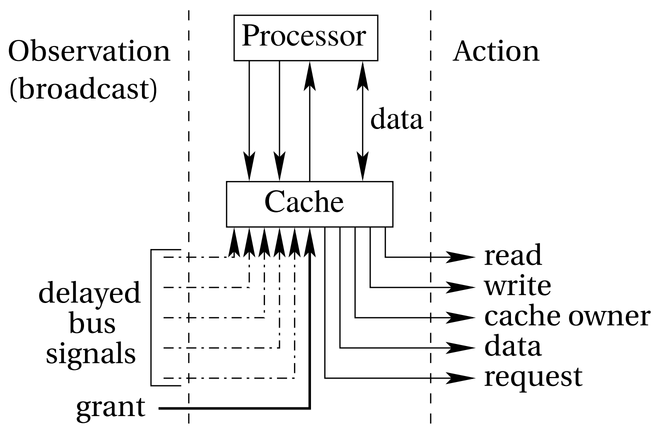


Figure 6.14: The cache controller and processor agent.

These components communicate using four signals: two for the processor to issue requests, one for the cache controller to signal completion, and a bidirectional data line. The Kesterel code for the processor is shown in Figure 6.15 and for the cache controller KBP in Figure 6.16.

The processors make an infinite sequence of unfair non-deterministic choices between doing local computation, or issuing read or write requests to the cache. (It is not correct to make these choices fairly as we have no information about what the processors are doing.) For memory operations the requests are sustained until the cache controller indicates that the operation has been completed.

Our most complex Kesterel process is the cache controller KBP. A cache uses a register to track its ownership of the cache line, which we have identified with dirtiness. A second register maintains the value of the line whenever the cache owns it.

```
processor :: (EC (⇝), ArrowNonDet (⇝) (B (⇝)), ArrowProbe (⇝) (B (⇝)))
          ⇒ Integer → (Signal, Signal, Signal, Signal) → E (⇝) () ()
processor i (proc_read, proc_write, proc_val, proc_cont) = loopE (procOp ⋙ pauseE)
  where
    nondetEL = foldr1 nondetE
    cacheOp op = loopE_ (λexn → op ⋙ whenE proc_cont (throwE exn) ⋙ pauseE)
    procOp = nondetEL
      [ nothingE   — Local operation
      , cacheOp (emitE proc_read)   — Memory operation: read
      , cacheOp (emitE proc_write)   — Memory operation: write 0
      , cacheOp (emitE proc_write ⋙ emitE proc_val) ]   — Memory operation: write 1
```

Figure 6.15: The Kesterel code for the processors.

The local signals $cache_{knows}$ and $cache_{knowsVal}$ track whether a cache knows the line, and if so, what its value is; the function cKnows is responsible for their maintenance at all times. Intuitively the knowledge conditional says that cache i knows the value of the line if a cache owns it and i knows its local value, or there is no owner and i knows the memory value. The constant *caches* is an enumeration of cache controller names.

The function cOwner responds to memory read requests when cache i owns the line. Responding to such requests has the effect of cleaning the cache. We avoid causality issues by using a delayed register to track dirtiness.

The cache protocol itself is a simplification of the Write-Once protocol: if a cache is clean then a write action involves a bus write that notionally claims ownership of the line; thereafter read and write requests can be satisfied locally. A read miss also involves a bus operation, though we allow a read to be satisfied by some other cache's read operation; specifically, if the cache becomes aware of the line's value while waiting to gain access to the bus, then the wait is aborted and the known value returned. (The call sustainWhileE *sig f* combinator emits *sig* while control resides in *f*; if *f* never pauses then *sig* is not emitted. We provide this as a primitive to avoid introducing extra state.)

This use of knowledge conditionals allows us to abstract from how the cache learns the line's value, and is guaranteed to make optimal use of knowledge with respect to read operations but not writes. We discuss other possible uses for knowledge conditionals in the following sections.

The reader will note that we have omitted the third type of operation in a cache protocol: discarding the line, either by simply forgetting it or flushing it to memory if we own it. We discuss why this operation is difficult to support in the following sections.

The system

The top-level system declaration composes the components we have described earlier: processors are composed in parallel with their cache controllers, and the resulting agent processes

```
cache i    — cache number
  (cache_owner, mem_read, mem_write, mem_val)    — bus signals
  bus_arb    — bus arbitration
  (proc_read, proc_write, proc_val, proc_cont)    — processor signals
  = registerE ("cval" ++ show i) $ λ(cLine_set, cLine_reset, cLine) →
    register_delayedE ("dirty" ++ show i) $ λ(dirty, clean, isDirty) →
    signalE $ λ(cache_knows, cache_knowsVal) →
        each_instantE [ cKnows i cache_knows cache_knowsVal
                      , cOwner cache_owner mem_read mem_val cLine clean isDirty ]
      ||||
        loopE (sigCaseE
          [ proc_read :- unlessE cache_knows
            (guardedBusOp bus_arb cache_knows (    — Read miss
              emitE mem_read ⋙ pauseE ⋙ pauseE))
            ⋙ whenE cache_knowsVal (emitE proc_val)
            ⋙ emitE proc_cont
          , proc_write :- unlessE isDirty
            (busOp bus_arb (    — Write miss
              emitE mem_read ⋙ pauseE ⋙ pauseE
              ⋙ emitE dirty
              ⋙ emitE mem_write ⋙ whenE proc_val (emitE mem_val)))
            ⋙ presentE proc_val (emitE cLine_set) (emitE cLine_reset)
            ⋙ emitE proc_cont ] ⋙ pauseE)
  where
    each_instantE = loopE ∘ foldr (⋙) pauseE
    busOp (req, ack) op = sustainWhileE req
      ((loopE_ $ λexn → whenE ack (throwE exn) ⋙ pauseE) ⋙ op)
    guardedBusOp (req, ack) cond op = sustainWhileE req
      (loopE_ $ λexn → presentE cond (throwE exn) (whenE ack (op ⋙ throwE exn))
                        ⋙ pauseE)
```

```
cOwner cache_owner mem_read mem_val cLine clean isDirty = whenE mem_read
  (whenE isDirty (emitE cache_owner ⋙ whenE cLine (emitE mem_val) ⋙ emitE clean))
```

```
cKnows i cache_knows cache_knowsVal =
  kTestE (owner_val i (¬)) (emitE cache_knows)
  (kTestE (owner_val i id) (emitE cache_knows ⋙ emitE cache_knowsVal) nothingE)
  where
    owner_val i pol = ("cache" ++ show i) ‘knows‘
      (⋀_{j ← caches} [ probe ("dirty" ++ show j) ⟶ pol (probe ("cval" ++ show j)) ]
      ∧ (⋀_{j ← caches} [ ¬ (probe ("dirty" ++ show j)) ] ⟶ pol (probe "memory value")))
```

Figure 6.16: The Kesterel code for the cache controller processes.

```
system = arbitratedBus processes
  where
    cAgent i bus_sigs bus_arb =
      signalE (λproc_sigs → processor i proc_sigs |||| cache i bus_sigs bus_arb proc_sigs)
    processes bus_sigs =
      ( (mkSizedListf (λi → ("cache" ++ show i, cAgent i bus_sigs)))
                 `withLength` (undefined :: Caches)
      , memory bus_sigs)
```

Figure 6.17: The Kesterel code for the top level.

placed in an arbitrated broadcast setting. The type-level constant Caches defines the number of agents in the system. The mkSizedListf combinator was discussed in §6.4.1.

6.6.2 Verification

We firstly verified a battery of sanity properties of the model, such as exclusiveness of ownership, the possibility of completing memory operations without bus operations, liveness of the processors and memory, and that a cache always knows the value of the memory line when it completes a memory operation but can be ignorant of it at other times.

Our main correctness property is that the processors have a suitable view of memory. In general memory consistency is difficult to specify and somewhat subjective; it is now common for higher-performance weak consistency models to push some of the problem back into software by loosening the ordering of reads and writes as viewed by different processors. Steinke and Nutt (2004) develop a theory that accounts for many consistency models.

In this case we can show *sequential memory consistency*, a concept due to Lamport (1979). According to Alur, McMillan, and Peled (2000, §2.2):

> The intuition behind sequential consistency ... is that an implementation of a collection of concurrent objects should appear to be correct to an observer that is able to record the history of each individual process, but has no global clock by which to determine the relative order of events of different processes.

We might say that in general consistency need only respect causality (§4.3.1).

Alur et al. (2000) go on to argue that verifying the sequential consistency of a finite-state system is undecidable, and conclude that each sequentially-consistent finite-state system obeys some stronger property. In our model we have the very strong property that a processor always reads the value most-recently written by any processor. We break this assertion into two cases, one for each possible value of the line. In the positive case we have:

$$\begin{aligned}
&\mathsf{AG}\ (\textstyle\bigwedge_{i \leftarrow caches}\ proc_{write}\ i \wedge proc_{val}\ i \wedge proc_{cont}\ i \\
&\qquad\qquad \longrightarrow\ \mathsf{AX}\ (\mathsf{A}[(\textstyle\bigwedge_{j \leftarrow caches}\ proc_{read}\ j \wedge proc_{cont}\ j \longrightarrow\ proc_{val}\ j) \\
&\qquad\qquad\qquad\qquad \mathsf{U}\ (\textstyle\bigvee_{j \leftarrow caches}\ proc_{write}\ j \wedge \neg\, proc_{val}\ j \wedge proc_{cont}\ j)\ ]))
\end{aligned}$$

and similarly for the negative case. Intuitively we have that, after completing a write of a one, in the next state all processors read a one until some processor completes the writing of a zero. We use an *unbounded* semantics for the until modality; it may be that the processors stop writing to memory at some point, and so we do not require the standard eventuality condition.

It is also the case that if a clean cache knows the value of the line then all caches do, and it is this property that prevents us from modelling the *exclusive unmodified* (E) state in the MOESI classification. In fact we can show that the memory register value is always common knowledge to all the caches, which we demonstrate by adding the test kTest ($\widehat{\mathsf{cknows}}_{caches}$ `"memory value"`) to one of the caches and verifying that it is always true. This result depends crucially on perfect recall as caches do not record the state of the line in their local states unless they are the owner. Thus it does not hold under the *observational* semantics for knowledge (§2.3.2).

This latter property is not surprising as the central memory simply records what everyone sees. However it also shows that this component is redundant in this model, which is clearly not true of any realistic shared memory system. We discuss this issue and the closely related problem of modelling cache flushes in the next section.

6.6.3 Concluding remarks

In contrast to the parametrised, compositional model we have described here, Baukus and van der Meyden (2004) manually expand the asynchronous composition of the memory process and two cache controllers. This is necessary as the modelling language of MCK lacks facilities for describing asynchronous systems, and the complexity of the resulting artifact makes it difficult to see that it is correct. For example, in their MCK models the `{Copy Back}` clause that cleans a dirty cache by writing the line back to memory has that cache and the central memory communicating without the other cache making an observation. This violates our assumption that bus communication is a broadcast. As a result the value in the central memory is not commonly known as the `{Copy Back}` clause is always enabled when a cache is dirty.

This part of their model also illustrates the cache flushing problem that we discussed earlier. In particular, when a cache controller flushes the line back to the memory we expect it to be forgotten. The knowledge-based specification of Baukus and van der Meyden (2004, §5) requires that a cache reset the variable it uses to track the line but not that it forget the value; that the cache *does* forget the value relies on the use of the observational semantics for the knowledge post-condition in their `{Read Miss}` clause, which is oblivious to history. In contrast the perfect-recall semantics we use here does not allow an agent to voluntarily forget anything, and so we cannot treat this facet of the protocol.

We also note that if the `{Copy Back}` action is broadcast in their model then the caches retain (perfect-recall) knowledge of the line after `{Copy Back}` and `{Flush}` operations. In the case of the Write-Once protocol we consider here, two consecutive bus writes by cache i indicates a `{Copy Back}` has occurred, and hence i has transferred ownership to the central memory, whose value is then commonly known. We conclude that their completeness result for this case hinges on an improper modelling of the bus. Indeed, if one does treat the bus as a broadcast then the asynchronous semantics for knowledge coincides with the synchronous one in our model.

A proper treatment of flushing requires us to account for the motivation of this operation: to recycle the space for another cache line. We conclude that perfect recall is not the right semantics for this task as it yields implementations with too many states; in practice cache protocols trade communication for space, whereas perfect recall favours memory over communication. One could imagine instead using the clock semantics and then model checking the implementation for perfect recall, but it is unclear this has any benefits over a standard model. Other options include making our KBP formalism space-aware, or adding a forgetting operation. We leave further exploration to future work.

One may wonder if our assumption that the system is globally synchronous limits the applicability of our implementations. We defend it by observing that we are modelling a single bus that all of the cache controllers are synchronised to; the processors can proceed at their own rate in some other clock domain or even asynchronously, but must synchronise with their controllers for memory operations. As none of this has any impact on the cache coherency protocol used on the bus, we can disregard it. Alglave, Maranget, Sarkar, and Sewell (2010) further argue that while such an assumption may not be entirely correct, it is adequate to capture the main ideas of even quite complex memory models.

We note that symmetry reduction (Cohen et al. 2009) may be a large win in this setting.

Extending this approach to hierarchical cache coherency protocols (Clarke et al. 1995) that include *bus bridges* would require some new theory as these violate our broadcast assumption.

6.7 Concluding remarks

Here we have described how we augment the circuit Arrows of the previous chapter with constructs for knowledge-based programming, and shown how we can implement the algorithms of Chapter 3 symbolically. We have applied these techniques to several examples, including that of cache coherency on a shared bus.

The following chapter sums up our experience.

Chapter 7

Conclusions and future work

We set out to convince the reader that mechanically reasoning about knowledge is useful when designing some kinds of systems. To that end we presented a theory of the implementation of knowledge-based programs in Chapter 3 that underpinned the symbolic approach shown in Chapters 5 and 6, which drew on the tradition of modelling circuits as functional programs that we surveyed in Chapter 4. Here we review our experience of using Arrows for KBPs and the KBP formalism itself, and point to future work.

7.1 Arrows for Knowledge-based Circuits

By embedding our modelling language in Haskell we have a superior foundation for experimentation to our previous MCK tool (§2.3.2). The new approach has much better support for data types. It is far easier to parametrise protocols and communication topologies and does not require recourse to another language to do so. Combinationally cyclic circuits (§4.1) ease composition and lead to smaller models than would otherwise be possible (§6.6). This greater flexibility leads to much better performance (§6.5) with the existing model checking algorithms. From a software engineering perspective the system itself is much more modular, maintainable and extensible, smaller and simpler, and we did not invest effort in the typical language processing drudgery of parsing, type checking, etc. We claim that the EDSL approach is the best way to build experimental language processors.

With respect to the tradition of describing circuits as functional programs that we surveyed in Chapter 4, Arrows have in a sense brought us back to the combinatory approach of μFP (§4.2.1) with the option of writing our definitions in pointwise or point-free styles (§5.1). By building the synchronous isomorphism into the Arrow structure (§4.3.5, §5.1.2) we have substantially avoided the explicit tuple spaghetti of μFP.

The reader may wonder if the conceptual and syntactic overhead of the techniques we use are necessary to resolve the issues we canvassed in Chapters 4 and 6. The following sections discuss the major facets of our approach and compare them to the alternatives.

7.1.1 The finally-tagless approach to "open" syntax

We specify our circuit Arrows using the "finally-tagless" approach (§5.1.2), which allows for an extensible (an "open") syntax in contrast to a deep embedding (i.e., syntax as an explicit datatype), and reinterpretation unlike a shallow embedding (i.e., no syntax at all). It also enables a treatment of circuits at different levels of abstraction (§5.4.1, §5.4.2). We note that it is not at all the same thing as an extensible semantics (§4.1).

There are several drawbacks to this approach, however. Firstly, the types and class contexts are often too complex to write and maintain. This is especially problematic when developing Arrow code as the Haskell standard requires us to write signatures to defeat the monomorphism restriction; this is one reason to remove that wart from the language. Another issue is that the implementation of an interpretation is scattered throughout the syntax class instances. This leads to much repetition – we must specify the class context of an instance and the types in its head in full – which makes it more difficult to identify and update invariants. Putting it another way, extensible syntax somewhat reduces the value of Haskell's compositional semantics.

Ambiguity must also be tamed somehow; we used *associated types* (also called *type families*) for this purpose (§5.2.1) but could just as easily have employed functional dependencies (Jones and Diatchki 2008). A drawback is that we can only support one type of Booleans per Arrow, which has not proven to be an issue in the examples we considered.

In the context of Arrows, our desire for reinterpretation has meant that we cannot avail ourselves of the **if – then – else** or **case** notation (§5.2.2, §5.3), and the **rec** construct can only be used for cycles that contain delays. This is more of an issue with Arrows than it would be without them as the syntax is already quite heavy. We might hope that the Arrow notation can be generalised; we discuss this further in §7.1.5.

Any approach to reinterpretation is open to the charge that the interpretations are not coherent, e.g., that the circuit we simulate or verify bears no relation to the netlist, or that a more abstract interpretation is consistent with a bit-level one. We can mitigate this somewhat by sharing the implementations of the bit-level operations amongst all the interpretations, but have no way of guaranteeing conformance.

7.1.2 Staging in EDSLs

In our setting Arrows are essentially a first-order combinatory language of circuits embedded in a higher-order language of circuit generators (i.e., Haskell). This allows us to insulate the semantics of the embedded (object) language from the host's (our meta-language), which we argue is necessary because the laws of the host are not those of circuits. In particular the host language is notionally sequential, whereas our circuits contain truly parallel operations that it cannot model (§4.1), and moreover circuits do not admit the same equations as the host. As Launchbury et al. (1999) observe, "silicon cannot be allocated on the fly".

This staging – producing Arrow graphs representing circuits by reducing Haskell expressions – is enforced by the type system, as we have a clear separation between functions that construct circuits and Arrows that represent them. We claim that it is therefore easier for users to understand this process of *elaboration* than in Lava 2000 (§4.2.5) where the two are conflated.

Unfortunately our staging is complicated by our use of both "finally-tagless" syntax and Arrows: in effect we now have two "compile times" – the first being the compilation of the Haskell source, and the second being the execution of the circuit generators – and therefore two sorts of "static" computation. As we argued in §5.3, types have the advantage of being relational, allowing information to flow between use context and a definition, but are a relatively weak language. A purer kind of staging would use a single expressive language for all static computation. See §4.4 for further discussion of such systems.

Another drawback of this approach is that we need many more structural combinators, indeed far more than even a Monadic version would require. In general we need to provide all sorts of higher-order machinery that Monads can freely reuse from the host language (due to their Cartesian closure, see §5.1), and moreover we often need variants of these to treat varying combinations of λ- and Arrow-bound arguments. Moreover many of these generalisations cannot be used in concert with the banana-brackets of the Arrow notation. For example, consider the mapSLn we used in our account of the Muddy Children puzzle in §6.4.1:

$$\begin{aligned}&\mathsf{mapSLn} :: (\mathsf{Arrow}\ (\rightsquigarrow), \mathsf{Card}\ \mathit{size})\\ &\qquad\Rightarrow (\mathsf{Integer} \to \alpha \rightsquigarrow \beta) \to \mathsf{SizedList}\ \mathit{size}\ \alpha \rightsquigarrow \mathsf{SizedList}\ \mathit{size}\ \beta\end{aligned}$$

Here we wish to pass a λ-bound *and* an Arrow-bound argument to the Arrow generator that we are mapping across the SizedList, which is not allowed. The position-oblivious version:

$$\mathsf{mapSL} :: (\mathsf{Arrow}\ (\rightsquigarrow), \mathsf{Card}\ \mathit{size}) \Rightarrow (\alpha \rightsquigarrow \beta) \to \mathsf{SizedList}\ \mathit{size}\ \alpha \rightsquigarrow \mathsf{SizedList}\ \mathit{size}\ \beta$$

does work with the notation however.

One might wonder how we can guarantee that the circuits are actually finite-state objects. In our setting we have this assurance for particular circuits if the constructivity interpretation of §5.4.3 converges, i.e., if it can produce a BDD encoding of the transition relation for a circuit. Krishnaswami, Benton, and Hoffmann (2012) have shown how to ensure that a discrete-time functional reactive program is finite state using linear types. It remains unclear that this style of programming is tractable, however, as a program effectively has to encode a finiteness proof in a way that the type system recognises. Perhaps it is better to split the logical correctness and space-use arguments as we do when designing synchronous systems such as circuits (§4.3.1).

7.1.3 Sharing in EDSLs

The problem of identifying shared expressions is common to many embedded languages and has been recently surveyed by Kiselyov (2011). The key difficulty is in treating recursive definitions, for a Monadic approach as employed by e.g. Xilinx Lava (§4.2.6) is sufficient when expressions are acyclic. DSLs that use the host language's notion of recursion must rely on impure features in the host language as there is no way to distinguish a recursive definition from its unrollings from within a pure language (§4.2.3). We conclude that preservation of equational reasoning in the meta language necessitates an explicit recursion combinator in the object language.

Arrows solve this problem by staging computations, as we discussed in the previous section, in such a way that the host language retains whatever laws it originally had. The **rec** notation is pleasant, when it applies, as it computes the tuple spaghetti involved in mutual recursion and inserts the recursion combinator. Unfortunately it is not general enough to support our operator for combinational cycles (§5.2.4), but as these are relatively rare, explicitly appealing to this combinator is not too burdensome.

7.1.4 Capturing information

One of our main motivations in using Arrows was to capture the observation an agent makes using the ArrowAgent class (§6.1). This is a similar problem faced by Li and Zdancewic (2010) who want to restrict the information available to a computation in a security setting: both settings require that *all* information sources for a particular Arrow are accounted for. They differ in that we also allow agents to make passive observations, such as the sensor readings made by our running example of the autonomous robot (Chapter 2).

Russo, Claessen, and Hughes (2008) showed that Monads can be used to a similar end, essentially by requiring that all relevant information comes from a Monadic action which tags the returned value, and testing that data being consumed has an appropriate tag. Our approach has the advantage that information producers can be oblivious to the needs of ArrowAgent, and we do not incur the extra overhead of testing tags at runtime. As Arrows naturally handle sharing, we also avoid the problem of identifying external inputs that the agent consumes several times, which simplifies the computation of agents' observations (§6.1). Moreover agents can make passive observations naturally by simply ignoring inputs provided by their environment.

We leave further investigation of this approach, and also the possibility of using observable sharing (§4.2.5), to future work.

7.1.5 Concluding remarks

Our approach leans heavily on the expressive types of modern Haskell systems, and it is difficult to imagine combinator programming without such a safety net. However the particular combinators that constitute Arrows have some limitations that suggest further research is worthwhile.

We note that the Arrow combinator graphs have a peculiar hybrid structure due to the arr operation, which effectively embeds the entire function space of the host language into the object language represented by the Arrows. This operation serves two roles: firstly, to interface the object language with Haskell's datatype mechanisms (case analysis, construction, etc.), and secondly to construct a limited set of pure Arrows that pipe data around, as we saw in §5.1. The generality of this mechanism makes it impossible to completely analyse Arrow graphs as we cannot divine the behaviour of a pure Arrow without executing it. The arr method is precisely the "polymorphic lift" that Carette et al. (2009) so scrupulously avoid.

This failure of the graph to completely represent the structure of the computation prevents us from generically transforming our circuits represented as Arrows. For instance we would like to implement a generic constant-folding Arrow transformer but are stymied by the impossibility of knowing what a pure Arrow does to our values. We could of course write such a transformation for each interpretation separately, but we would hope to do it once-and-for-all in a similar manner to the optimisations of Hughes (2004), which act structurally on the Arrow graph without reference to the semantics of what is passed between the combinators.

Another motivation for abstracting the pure Arrows used for routing is that some systems are better modelled with environments represented as sums rather than products; for instance Carlsson[1] experimented with an Arrow interface for his asynchronous fudget stream processors. This approach also requires the first method to be suitably renovated, and still allows the machinery of the host language for datatypes to be used in the embedded language.

A more general approach is being pursued by Megacz (2010, 2011) with his "Generalised Arrows", which isolate the host and object languages by only requiring the latter to support a particular (abstract) set of pure routing Arrows. This means that the object language does not need to support the datatypes of the host language, which is precisely what we want for our synchronous circuit Arrows; these do not have a natural encoding of arbitrary sum types, for example (§5.2.2). It occupies another point in the space of "box and arrow" description techniques (§4.3.4), requiring a novel modal type system and a notation distinct from the one used here.

Following our discussion of circuit semantics in §4.3.5, one may wonder if Arrows are useful when reasoning about circuits formally, using a proof assistant. We observe that it is difficult to deeply embed the Arrow language in a simply-typed logic such as Isabelle/HOL (Nipkow et al. 2002) as we cannot give them sufficiently polymorphic types. (We could model them using a closed universe of types and so forth, but this might entail formalising a sufficiently expressive type system as well. Hughes (2004) had similar difficulty in Haskell prior to the advent of *generalised algebraic data types* (GADTs) (Schrijvers, Jones, Sulzmann, and Vytiniotis 2009).) Note also that Arrows suffer from the same sort of over-specification as do Monads (§4.2.4), viz that they encode the order that gates are defined in, and hence are not fully abstract; we need to further assume that the bind operator ($\ggg$) is suitably commutative.

More tellingly we follow Tullsen (2002) in observing that we almost certainly want to take an extensional view of our circuits, at least when verifying their functional properties, and showing that a transformation preserves correctness. In other words, we would like the freedom to ignore sharing provided by a direct semantics based on the domain theory mechanised by Huffman

[1] http://www.carlssonia.org/ogi/ProdArrows/

(2012) (et al), which can model combinational cycles if we need them. Type theorists may prefer to show that their circuits yield productive coinductive definitions in Coq or Agda (§4.3.5).

This does not contradict our motivation for using Arrows within Haskell, as sharing there is a critical issue (§7.1.3). We acknowledge the temptation to abandon purity and adopt observable sharing (§4.2.5) if we are only going to reason informally about our circuit generators expressed in the host language.

7.2 Representations and implementation techniques

In Chapter 6 we developed a symbolic version of the algorithm for constructing implementations of knowledge-based programs that we developed in Chapter 3 and applied it to several examples. We observe that while our prototype performed sufficiently well to draw some conclusions about the KBP formalism (§7.3), there are several ways one might go about scaling it up to larger examples, and applying it to systems that do not satisfy the structural expectations of §3.7.

Firstly we acknowledge that there are many generic optimisations we could also use, such as reducing the intermediate BDDs while saturating relations (Geldenhuys and Valmari 2001), exploiting symmetries (Cohen et al. 2009), and adopting other representations of explicit-state automata (Valmari 1996). We could also refine the splitting of the temporal slice under the agents' observations (§6.2.1) by identifying functional dependencies between different parts of these observations (Hu and Dill 1993).

The approach we pursued here can be seen as providing termination proofs of the graph traversal that constructs the automaton, which rests on the existence of a suitable simulation (§3.6.4). A more general alternative is to interleave traversal and automata reduction in a way that is guaranteed to terminate iff an implementation exists. van der Meyden (1996c) presents a method based on this idea that terminates iff there exist finite-state implementations of a set of KBPs with respect to the perfect-recall semantics for knowledge in a given synchronous scenario (§3.7.2). This approach essentially labels the states of the automata being constructed with the entire Kripke structure for a temporal slice, paired with a trace. As the search space is infinite, and both state-space traversal and bisimulation reduction alone yield infinite reachable state spaces, these operations must be interleaved for the method to terminate. In our setting the simulations are also used to optimise the representation of the automaton states (§3.6.4). This is essential to any attempt to construct implementations in feasible amounts of memory.

That interleaving minimisation and generation can yield finite-state systems where neither does alone was observed by Lee and Yannakakis (1992, §3). A contemporary approach by Bouajjani, Fernandez, Halbwachs, and Raymond (1992) essentially fuses the bisimulation reduction and reachability algorithms in that order, and is therefore limited to systems where the number of equivalence classes of the state space under bisimulation is finite. They also observe that the complexity arguments of Lee and Yannakakis (1992) are misleading as the costs of the BDD operations dominate all others, and these are difficult to predict.

We note that while bisimulation reduction is sufficient to guarantee termination under the conditions given by van der Meyden (1996c), we really want a stronger form of minimisation, as we discussed in §6.2.4.

As we mentioned in §6.1, developing an explicit-state version of this framework may be worthwhile as it could prove more efficient for some data-oriented scenarios where BDDs explode, such as the Mr. S and Mr. P puzzle of §6.4.2. We leave the investigation of this and other representations to future work.

One alternative to these exact approaches is to adapt a learning algorithm and heuristic minimisation following Chen, Farzan, Clarke, Tsay, and Wang (2009). In this setting we would lazily refine implementation automata against the KBP scenario until the system satisfied a property (typically weaker than the implementation relation of §3.6.2) specified by the designer. Such a scheme would allow for potentially smaller implementations that are less perfectly knowledgeable. A semantics for this approach might adopt the *local propositions* of Engelhardt et al. (2001).

An interactive interpreter for the KBP formalism would be very useful when debugging specifications appealing to knowledge.

7.3 The KBP formalism

We presented several knowledge-based programs in Chapter 6, and here review the formalism itself. As we have presented it, KBPs are a programming formalism where guards are allowed to contain explicit tests for knowledge. This is something of a pleasant combination as we can conveniently use programming constructs such as sequential composition and datatypes rather than having to encode them into a logic, and knowledge operators are useful for making inferences about state that agents cannot directly observe.

This formalism is a stylised subset of the full synthesis task (van der Meyden and Vardi 1998) where the behaviour of the agents is specified using linear temporal logic (LTL) augmented with epistemic modalities. This task is computationally intractable even when restricted to the LTL sub-language (Pnueli and Rosner 1989). We might argue that pure logic is not an ideal starting point for synthesis as it lacks the programming constructs that even specifications require, which can lead to the specification being less comprehensible than the implementation. Moreover one must somehow indicate the system architecture to such tools to avoid it producing trivial centralised solutions (Wolper 1998). We contend that the KBP formalism sketched here is a decent compromise: we specify the system architecture and automatically analyse the information flow between the components. The automation helps greatly with exploring the epistemic properties of the system being designed.

As we discussed in §6.4.2, the KBP formalism might usefully be extended with temporal operators, with the aspiration that constructing implementations remains possible. We showed there that the existing approach is easily extended with *past-time* operators if these are used in a suitably restricted way. The full combination requires adding propositions to the state, which can be done as a pre-processing step; the algorithm does not require adjustment.

Much more difficult is to add *future-time* operators as these would require a consideration of infinitary behavior that is beyond the inductive construction presented here. A promising direction for future research is to integrate the recent work of Piterman, Pnueli, and Sa'ar (2006) (etc.) for sub-languages of LTL into a KBP formalism.

A limitation of the KBP formalism we use here is that we need to specify exactly what actions to perform. For this reason we cannot give an interesting treatment of the bit transmission problem following Halpern and Zuck (1992); while they do present synchronous implementations of their KBP, such as the AUY protocols (Aho, Ullman, and Yannakakis 1979), the relation between specification and implementation involves *action refinement*, where an action in the KBP is realised as several steps in the implementation. It is beyond the reach of our tool to perform this step automatically, and we found that the specification is far too concrete if we perform it manually.

We conclude with some suggestions for future application domains. Distributed protocols for termination, garbage collection and mutual exclusion all rely on agents reasoning about incompletely observed state, as does the scheme for matching hardware bus protocols proposed by Avnit, D'Silva, Sowmya, Ramesh, and Parameswaran (2009). The synthesis of fault-tolerant discrete controllers may benefit from a treatment of unobservable state offered by an epistemic formalism (Attie, Arora, and Emerson 2004; Girault and Rutten 2009). Wireless networks are a setting where computation is far cheaper than communication, so making maximal use of information is important; the perfect-recall semantics may prove useful here. Vasudevan, DeCleene, Immerman, Kurose, and Towsley (2003) develop a leadership election protocol for such networks, starting with a synchronous design that is mapped to an asynchronous implementation. KBPs may prove useful in the first of these steps, as we suggested in §1.1.

Appendices

Appendix A

Model Checking Knowledge and Linear Time: PSPACE Cases

With Kai Engelhardt and Ron van der Meyden, I showed some complexity results for the related problem of model checking systems using linear temporal logic and knowledge (Engelhardt et al. 2007). It provides another example of using simulations to reduce the redundancy in Kripke structures along the lines of Chapter 3. I contributed to the proofs of the results in §§A.4-A.6.

This appendix contains the paper as published with complete proofs in line.

Abstract We present a general algorithm scheme for model checking logics of knowledge, common knowledge and linear time, based on bisimulations to a class of structures that capture the way that agents update their knowledge. We show that the scheme leads to PSPACE implementations of model checking the logic of knowledge and linear time in several special cases: perfect recall systems with a single agent or in which all communication is by synchronous broadcast, and systems in which knowledge is interpreted using either the agents' current observation only or its current observation and clock value. In all these results, common knowledge operators may be included in the language. Matching lower bounds are provided, and it is shown that although the complexity bound matches the PSPACE complexity of the linear time temporal logic LTL, as a function of the model size the problems considered have a higher complexity than LTL.

A.1 Introduction

The logic of knowledge (Fagin et al. 1995) has been proposed as a formalism to express information theoretic properties in distributed and multi-agent systems, and has been shown to be useful for the analysis of distributed systems protocols (Halpern and Moses 1990), information flow security properties (Halpern and O'Neill 2003; Syverson 1992; van der Meyden and Su 2004), as well as for problems such as diagnosis and recoverability (Cimatti, Pecheur, and Cavada 2003; Cimatti, Pecheur, and Lomuscio 2005).

The semantics for knowledge operators can be defined in a variety of ways, depending on what information agents use when computing what they know. At one extreme (the "observational semantics") agents rely only on their current observation, at the other (the "synchronous perfect recall semantics") agents rely on the log of all their past observations. In between lies a "clock semantics" in which agents rely on their current observation plus a clock value. These semantics have different motivations: the perfect recall semantics is most appropriate for security analyses and derivation of protocols that make optimal use of information; the other semantics are closer to system implementations.

A number of model checkers for the logic of knowledge have been recently developed, which embody different choices of semantics for the knowledge operators and different types of expressiveness for the temporal dynamics. MCMAS (Lomuscio and Raimondi 2006b) deals with the observational interpretation of knowledge and the branching time logic CTL. DEMO (van Eijck 2007) deals with the dynamic logic based "update logic" (Baltag and Moss 2004), which handles what is in effect the perfect recall semantics for knowledge. The system MCK (Gammie and van der Meyden 2004) covers a broad spectrum of definitions of knowledge (observational, clock, perfect recall), as well as dealing with both linear time and branching time temporal logic.

Where they deal with the perfect recall semantics for knowledge, these systems place severe constraints on the interaction between knowledge and temporal operators, for reasons of inherent complexity. The complexity of model checking the combination of the linear time temporal logic LTL with knowledge operators interpreted according to the perfect recall semantics has been studied by van der Meyden and Shilov (van der Meyden and Shilov 1999), who show that this problem is decidable but with a non-elementary lower bound, and undecidable when operators for common knowledge (a type of fixed point over knowledge operators) are added to the language. (Shilov et al (Shilov and Garanina 2002, 2006; Shilov, Garanina, and Choe 2006) have also studied branching time versions of these results.)

However, as we show in this paper, this general result does not preclude the existence of special cases in which this model checking problem has lower complexity, even when common knowledge operators are included in the language. We identify a number of cases where the problem (including common knowledge) is solvable in PSPACE. These include systems with a single agent (discussed in Section A.5.1) and systems in which all communication is by synchronous broadcast (treated in Section A.5.2). The result concerning a single agent improves the nonelementary upper bound for the single agent case obtained from the algorithm of van der Meyden and Shilov.

Our approach to the proof of these results is by means of a general algorithm scheme (presented in Section A.4) that relies upon the existence of a bisimulation from the (in effect, infinite) systems being checked to a finite structure that represents the way that agents update their knowledge in the system. In addition to the results about the perfect recall semantics, we show that this scheme can be used to obtain PSPACE complexity results for model checking the logic of knowledge and linear time for other interpretations for knowledge: in particular, we show that this complexity bound applies in the case of both the clock semantics and the observational semantics (see Section A.6).

As the complexity of model checking the linear time temporal logic LTL alone is already PSPACE-complete, it may seem from these results that the extra expressiveness of the logic of knowledge in these cases comes at no extra cost. In fact, we show that there is a sense in which these model checking problems are harder than model checking LTL alone, by focussing on the complexity of model checking a fixed formula as a function of the size of the model. For LTL, this "model complexity" is linear-time for each formula (Lichtenstein and Pnueli 1985). We show that the model complexity can be as high as PSPACE-complete once the formula includes knowledge operators.

A.2 Basic definitions

In this section we define the semantic framework with respect to which we study the model checking problem. The definitions closely follow (van der Meyden and Shilov 1999), which dealt with model checking knowledge and linear time in multi-agent systems for a "perfect recall" interpretation of knowledge. We also define an alternate "clock" interpretation of knowledge, in which agents reason on the basis of their current observation and knowledge of the time.

Let *Prop* be a set of atomic propositional constants, $n > 0$ be a natural number, and let $\mathbb{A} = \{1, \ldots, n\}$ be a set of agents. We will be concerned with model checking a propositional multi-modal language for knowledge and linear time based on the set *Prop* of atomic propositional constants, with formulas generated by the modalities $\bigcirc$ (next), $\mathcal{U}$ (until), a knowledge operator K_i for each agent $i \in \mathbb{A}$, and a common knowledge operator C_G for each group of agents $G \subseteq \mathbb{A}$. Formulas of the language are defined as follows: each atomic propositional constant $p \in Prop$ is a formula, and if φ and ψ are formulas, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\bigcirc\varphi$, $\varphi\,\mathcal{U}\,\psi$, $K_i\varphi$ and $C_G\varphi$ for each $i \in \mathbb{A}$ and group $G \subseteq \mathbb{A}$. We write $\mathcal{L}_{\{\bigcirc,\mathcal{U},K_1,\ldots,K_n,C\}}$ for the set of formulas. We will refer to sublanguages of this language by a similar expression that lists the operators generating the language. For example, $\mathcal{L}_{\{\bigcirc,\mathcal{U},K\}}$ refers to the sublanguage with just a single agent (in which case we may drop the subscript on the knowledge operator). As usual in temporal logic, we use the abbreviations $\Diamond\varphi$ for $\mathbf{true}\,\mathcal{U}\,\varphi$, and $\Box\varphi$ for $\neg\Diamond\neg\varphi$. The *knowledge depth* of a formula φ, denoted $depth(\varphi)$, is defined to be the maximal depth of nesting of K operators in φ. For example, $depth(K(p \wedge \neg Kq)) = 2$.

The semantics of this language is defined with respect to the following class of structures. Define an *interpreted environment (for $\mathbb{A}$)* to be a tuple E of the form $(S, I, \rightarrow, (O_i)_{i\in\mathbb{A}}, \pi, \alpha)$ where the components are as follows:

1. S is a set of *states* of the environment,
2. I is a subset of S, representing the possible *initial states*,
3. $\rightarrow \subseteq S^2$ is a *transition relation*,

4. for each $i \in \mathbb{A}$ the component $O_i : S \longrightarrow \mathcal{O}$, where $\mathcal{O}$ is a set of uninterpreted observations, is called the *observation function of agent* i,
5. $\pi : S \longrightarrow \mathcal{P}(Prop)$ is an *interpretation*,
6. $\alpha \subseteq S$ is an *acceptance condition*.

Intuitively, an environment is a transition system where states encode values of local variables, messages in transit, failure of components, etc. For states s, s' the relation $s \rightarrow s'$ means that if the system is in state s, then at the next tick of the clock it could be in state s'. We call E finite whenever S is. If s is a state and i an agent then $O_i(s)$ represents the observation agent i makes when the system is in state s, i.e., the information about the state that is accessible to the agent. The interpretation π maps a state s to the set of propositional constants in *Prop* that hold at s. The acceptance conditions are essentially Büchi conditions which model fairness requirements on evolutions of the environment.

A *path* p of E from a state s in S is a finite or infinite sequence of states $s_0 s_1 \dots$ such that $s_0 = s$ and $s_j \rightarrow s_{j+1}$ for all j. We write $p(m)$ for s_m when m is an index of p. A path p is said to be *initialized* if $p(0) \in I$. We call an initialized finite path a *trace*. A path p is *fair* if it is infinite and $p(i) \in \alpha$ for infinitely many i. Note that we do not assume that S is finite, but when so, this formulation is equivalent to the usual formulation of acceptance for Büchi automata: some $s \in \alpha$ occurs infinitely often. We say that the acceptance condition of E is *trivial* if $\alpha = S$. We assume that environments satisfy the following well-formedness condition: for every state s, there exists a fair path with initial state s. A *run* of E is a fair, initialized path, and we write $r[0..m]$ for the trace that is the prefix of run r up to time m. Let *runs*(E) be the set of all runs of E. A *point* of E is a pair (r, m), where r is a run of E and m a natural number. Intuitively, a point identifies a particular moment in time along the history described by the run.

Individual runs of an environment provide sufficient structure for the interpretation of formulas of linear temporal logic. To interpret formulas involving knowledge, we use the agents' observations to determine the points they consider possible. There are many ways one could do this. The particular approaches used in this paper model a *synchronous perfect-recall*, an *observational*, and a *clock* semantics of knowledge, each defined using a notion of local state. We define the *synchronous perfect recall local state of agent* i *at a point* (r, m) to be the sequence[1] $\{(r, m)\}_i^{\mathtt{pr}} = O_i(r[0..m])$. That is, the synchronous perfect recall local state of an agent at a point in a run consists of a complete record of the observations the agent has made up to that point. The *clock local state of agent* i *at a point* (r, m) is defined by $\{(r, m)\}_i^{\mathtt{clk}} = (m, O_i(r(m)))$. That is, in this definition, the agent's local state is taken to be the current time, together with the agent's current observation. Finally, the *observational local state of agent* i *at a point* (r, m) is $\{(r, m)\}_i^{\mathtt{obs}} = O_i(r(m))$. Effectively, an agent with this view of the world considers any reachable state giving the same observation to be possible. To distinguish these local state assignments, we define a *view* v to be one of the three possibilities `pr`, `clk`, and `obs`.

[1] We adopt the convention that functions lift to sequences and sets in a pointwise fashion.

Given a view v, the corresponding local state assignment may be used to define for each agent i a relation $\stackrel{v}{\sim}_i$ of *indistinguishability* on points $(r,m),(r',m')$ of E, by $(r,m)\stackrel{v}{\sim}_i(r',m')$ if $\{(r,m)\}_i^v = \{(r',m')\}_i^v$. Intuitively, when $(r,m)\stackrel{v}{\sim}_i(r',m')$, agent i's local state according to the view v does not contain enough information for the agent to determine whether it is at one point or the other. Clearly, each $\stackrel{v}{\sim}_i$ is an equivalence relation. Both the synchronous perfect recall view and the clock view are "synchronous" in the sense that if $(r,m)\stackrel{v}{\sim}_i(r',m')$, then we must have $m=m'$. Intuitively, this means that the agent "knows the time". The relations $\stackrel{v}{\sim}_i$ will be used to define the semantics of knowledge for individual agents. By $P_i^v(E,r,m)$ we denote the set $\{\, r'(m') \mid r'\in runs(E),\ m'\in\mathbb{N},\ (r',m')\stackrel{v}{\sim}_i(r,m)\}$ of *possible states for agent i* at point (r,m).

To interpret the common knowledge operators, we use another relation. If $G\subseteq\mathbb{A}$ is a *group* of agents (i.e., two or more) then we define the relation $\stackrel{v}{\sim}_G$ on points to be the reflexive transitive closure of the union of all indistinguishability relations $\stackrel{v}{\sim}_i$ for $i\in G$, i.e., $\stackrel{v}{\sim}_G = (\bigcup_{i\in G}\stackrel{v}{\sim}_i)^*$.

The semantics of this language is defined as follows. Suppose we are given an environment E with interpretation π. We define satisfaction of a formula φ at a point (r,m) of a run of E with respect to a view v, denoted $E,(r,m)\models^v\varphi$, inductively on the structure of φ. The cases for the temporal fragment of the language are standard, and independent of v:

$E,(r,m)\models^v p$	if $p\in\pi(r(m))$, where $p\in Prop$,
$E,(r,m)\models^v \varphi_1\wedge\varphi_2$	if $E,(r,m)\models^v\varphi_1$ and $E,(r,m)\models^v\varphi_2$,
$E,(r,m)\models^v \neg\varphi$	if not $E,(r,m)\models^v\varphi$,
$E,(r,m)\models^v \bigcirc\varphi$	if $E,(r,m+1)\models^v\varphi$,
$E,(r,m)\models^v \varphi_1\,\mathcal{U}\,\varphi_2$	if there exists $m''\geq m$ such that $E,(r,m'')\models^v\varphi_2$ and $E,(r,m')\models^v\varphi_1$ for all m' with $m\leq m'<m''$.

The semantics of the knowledge and common knowledge operators is defined by:

$E,(r,m)\models^v K_i\varphi$	if $E,(r',m')\models^v\varphi$ for all points (r',m') of E satisfying $(r',m')\stackrel{v}{\sim}_i(r,m)$
$E,(r,m)\models^v C_G\varphi$	if $E,(r',m')\models^v\varphi$ for all points (r',m') of E satisfying $(r',m')\stackrel{v}{\sim}_G(r,m)$

These definitions can be viewed as an instance of the "interpreted systems" framework for the semantics of the logic of knowledge proposed in (Halpern and Moses 1990). Intuitively, an agent knows a formula to be true if this formula holds at all points that the agent is unable to distinguish from the actual point. Common knowledge may be understood as follows. For G a group of agents, define the operator E_G, read "everyone in G knows" by $E_G\varphi\equiv\bigwedge_{i\in G}K_i\varphi$. Then $C_G\varphi$ is equivalent to the infinite conjunction of the formulas $E_G^k\varphi$ for $k\geq 1$. That is, φ is common knowledge if everyone knows φ, everyone knows that everyone knows φ, etc. We refer the reader to (Fagin et al. 1995) for further motivation and background.

A.3 Main results

We may now define the model checking problem we consider in this paper and state our main results.

Say that formula φ is *realized* in the environment E with respect to a view v, denoted $E \models^v \varphi$, if, for all runs r of E, we have $E,(r,0) \models^v \varphi$. Our interest is in the following problem, which we call the *realization problem* with respect to a view v: given an an environment E and a formula φ of a language $\mathscr{L}$, determine if φ is realized in E with respect to v.

The realization problem for the logic of knowledge and linear time has been studied by van der Meyden and Shilov (van der Meyden and Shilov 1999), who show that for the perfect recall view, the problem is undecidable for the language $\mathscr{L}_{\{\bigcirc,\mathscr{U},K_1,\ldots,K_n,C\}}$ when $n \geq 2$, and decidable for the language $\mathscr{L}_{\{\bigcirc,\mathscr{U},K_1,\ldots,K_n\}}$, but with nonelementary complexity. More specifically, for $\mathscr{L}_{\{\bigcirc,\mathscr{U},K_1,\ldots,K_n\}}$ their approach runs in space polynomial in $f(depth(\varphi), O(|E|))$, where the function f is defined by $f(0,m) = m$ and $f(k+1,m) = 2^{f(k,m)}$. It is also shown by van der Meyden and Shilov that there is a similar lower bound on the complexity when there is more than one agent.

Our main contribution in this paper is to develop a general algorithm scheme for model checking the logic of knowledge and time based on a notion of bisimulation of environments, and to show that this scheme yields improved complexity bounds in a number of special cases. The scheme itself is presented in Section A.4, and parameterizes a procedure for model checking with respect to the observational view. In particular, this procedure yields the following result for the observational view.[2]

Theorem 1. *Determining if a given formula in the language $\mathscr{L}_{\{\bigcirc,\mathscr{U},K_1,\ldots,K_n,C\}}$ is realized in a given environment E with respect to the observational view is decidable in PSPACE.*

By showing the existence of bisimulation from an enviroment representing the perfect recall semantics for a single agent to a suitable finite environment, we obtain the following result:

Theorem 2. *Determining if a given formula in $\mathscr{L}_{\{\bigcirc,\mathscr{U},K\}}$ is realized in a given environment E with respect to the perfect recall view is in PSPACE.*

This shows that the complexity of the realization problem for formulas with a single agent with perfect recall is strictly lower than the general case, and significantly improves upon the complexity bound of van der Meyden and Shilov in this case.

By finding other suitable structures we may derive complexity bounds on several other cases of the realization problem, as stated in the following results. First, although with respect to the perfect recall view, the realization problem is non-elementary for $\mathscr{L}_{\{\bigcirc,\mathscr{U},K_1,\ldots,K_n\}}$, there exist classes of environments with respect to which the problem has lower complexity, even if we add the common knowledge operators. In particular, this holds for *broadcast environments* (van der

[2] This result does not appear to have been previously stated in the literature, but we note that results of Vardi (Vardi 1996) on the problem of verifying that a concrete protocol implements a knowledge-based program are very closely related. Lomuscio and Raimondi have studied the complexity of model checking the combination of the logic of knowledge with the branching time logic CTL with respect to the observational semantics (Lomuscio and Raimondi 2006a).

Meyden 1996b). Intuitively, these are environments in which the only communication mechanism available to agents is to broadcast to *all* agents in the system. The formal definition will be given in section A.5.2. For broadcast environments we show the following.

Theorem 3. *Determining if a given formula in the language $\mathcal{L}_{\{\bigcirc,\mathcal{U},K_1,\ldots,K_n,C\}}$ is realized in a given broadcast environment E with respect to the perfect recall view is decidable in PSPACE.*

Realization for the clock view may also handled using the bisimulation technique and again the common knowledge operator may be included in the language.

Theorem 4. *Determining if a given formula in the language $\mathcal{L}_{\{\bigcirc,\mathcal{U},K_1,\ldots,K_n,C\}}$ is realized in a given environment E with respect to the clock view is decidable in PSPACE.*

Note that the complexity of model checking linear time temporal logic (i.e. realization for the language $\mathcal{L}_{\{\bigcirc,\mathcal{U}\}}$) is PSPACE-complete (Sistla and Clarke 1985). Since $\mathcal{L}_{\{\bigcirc,\mathcal{U}\}}$ is a sublanguage of the languages in the above results, these results show that the above bounds are tight, in the sense that the problems are in fact PSPACE-complete.

That some of our complexity bounds are no more than the PSPACE complexity of the linear time temporal logic LTL may at first suggest that model checking these cases of the logic and knowledge and time could be as effective in practice as model checking LTL. However, a closer inspection indicates that it is not obvious that this will be case. The time complexity of LTL model checking a *fixed* formula is linear in the size of the model. The time complexity is exponential in the size of the formula. This exponential bound is not an impediment in practice since the formulas of interest tend to be small. The models, on the other hand, may be very large. We show that as a function of model size, the complexity of model checking fixed formulas of the logic of knowledge and time falling within our PSPACE cases can be be as high as PSPACE-hard (for $\mathcal{L}_{\{\bigcirc,\mathcal{U},K\}}$ with respect to perfect recall) and at any level of the polynomial hierarchy for the clock view.

Theorem 5. *There exists a formula φ of $\mathcal{L}_{\{\bigcirc,\mathcal{U},K\}}$ such that the problem of deciding if φ is realized in a given environment E with respect to the perfect recall view is PSPACE-hard.*

Proof. By reduction from the problem of deciding if, for a given nondeterministic finite state automaton A over an alphabet Σ, the language $L(A)$ is equal to the universal language Σ^*.

Let A = (Q, q_0, δ, F) be an NFA with states Q, initial state q_0, transition function $\delta : Q \times \Sigma \longrightarrow 2^Q$ and final states F. We define an environment E_A that has two different types of runs: one corresponds to the generation of a sequence of inputs to A, the other corresponds to runs of A. We employ the special letter $\epsilon \not\in \Sigma$ to handle the empty word in both types of runs. To ensure that E_A has a fair path starting at every state, we add the sink state $\perp \not\in \Sigma$. Formally, the environment $E_A = (S, I, \rightarrow, O_1, \pi, \alpha)$ consists of:

- states $S = \Sigma \cup \{\epsilon, (\epsilon, q_0), \perp\} \cup \Sigma \times \Sigma$,

- initial states $I = \{\epsilon, (\epsilon, q_0)\}$,
- transitions
  - $\epsilon \to l$ and $l \to l'$ for each $l, l' \in \Sigma$,
  - $(l, q) \to (l', q')$ for each $l \in \Sigma \cup \{\epsilon\}, q \in Q, l' \in \Sigma$ and $q' \in \delta(q, l')$,
  - $(l, q) \to \perp$ if $\delta(q, l) = \emptyset$,
  - $\perp \to \perp$,
- observation function $O_1(l) = l = O_1(l, q)$, for all $l \in \Sigma \cup \{\epsilon\}$ and $q \in Q$, and $O_1(\perp) = \perp$,
- interpretation π given by
  - $\pi(\perp) = \emptyset$,
  - $\pi(l) = \{\mathtt{in}\}$ for $l \in \Sigma \cup \{\epsilon\}$,
  - $\pi(l, q) = \{\mathtt{final}\}$ if $l \in \Sigma \cup \{\epsilon\}$ and $q \in F$ else $\pi(l, q) = \emptyset$, and
- trivial acceptance condition α.

Note that for $w \in \Sigma^*$ of length m, if $r[0\cdots] = \epsilon.w$ then $P_l^{\mathtt{pr}}(E_A, r, m) = \{l\} \cup \{(l, q) \mid q_0 \to^w q\}$, where $l = r(m)$. Since there is such a run r for every word $w \in \Sigma^*$, it follows that $E_A \models^{\mathtt{pr}} \Box(\mathtt{in} \Rightarrow \neg K \neg \mathtt{final})$ iff $L(A) = \Sigma^*$. $\Box$

In the case of the clock semantics, we may obtain the following lower bound.

Theorem 6. *For each level Π_k^p of the polynomial hierarchy, there exists a formula φ of the language $\mathscr{L}_{\{\bigcirc, \mathscr{U}, K_1, \ldots, K_n\}}$ such that the problem of deciding, given an environment E, whether $E \models^{\mathtt{clk}} \varphi$, is Π_k^p-hard.*

Note that this implies PSPACE-hardness of the version of the problem in which the formula is given.

Proof. Fix k, and consider Π_k^p quantified Boolean formulas Φ of the form

$$\forall q_1^k \ldots q_n^k \exists q_{q_1}^{k-1} \ldots q_n^{k-1} \ldots (\forall/\exists) q_1^1 \ldots q_n^1 (\alpha),$$

where α is a 3-CNF formula of propositional logic in the variables q_i^j. (Formulas with differing numbers of propositional variables in the quantifications can always be put into this form at polynomial cost $O(nk)$ symbols by padding with unused variables.)

We construct environments E corresponding to such formulas in which the transition relation is the disjoint union of cycles of the form $s_0 \to \cdots \to s_{N-1} \to s_0$ We call such a component of the transition relation a *cycle of length N*.

Such cycles are used to represent assignments to the truth values of the propositional variables q_i^j as follows. Let $p_1^1 \ldots p_n^1, \ldots, p_1^k \ldots p_n^k$ be the sequence of the first nk primes greater than 2.

Then the largest number p_n^k in this sequence is known to be $O(nk(\log nk + \log\log nk)) = O(n^2)$. Let $N_i^j = \Pi_{1\leq j'\leq j} p_i^{j'}$. Thus the largest of these numbers is $N_n^k = O(n^{2k})$.

We associate with each variable q_i^j several cycles, each of length N_i^j, with one such cycle for each positive or negative occurrence of q_i^j in α. Let $\alpha = \bigwedge_{c\in C} c$, where each $c = \{l_1^c, l_2^c, l_3^c\}$ is a set representing a disjunction of 3 literals. If q_i^j or $\neg q_i^j$ occurs in c, we include in E a cycle $x_0^{c,i,j} \to \dots x_N^{c,i,j} \to x_0^{c,i,j}$ where $N = N_i^j - 1$. Note that occurrences of a variable in distinct clauses give rise to distinct cycles, i.e., if $c \neq c'$ then $x_l^{c,i,j} \neq x_m^{c',i,j}$, but these cycles have the same length. Of these states, the states $x_0^{c,i,j}$ are made initial. Thus we have one initial state per cycle in the transition relation. The total number of states is $O(|\Phi| \cdot N_n^k) = O(|\Phi|^{2k+1})$.

We make all the states arising from the clause c mutually indistinguishable to agent 1, i.e., we define $O_1(x_l^{c,i,j}) = c$. The observation function for agent 2 is defined so as to make all states indistinguishable, i.e., $O_2(x) = \perp$ for all states x.

Let X^j be the set of states $x_l^{c,i,j}$, and call these the *level l* states. It follows that if $P_m = P(E, r, m)$ is the set of states possible at time m, then

$$P_m \cap X^j = \{x_l^{c,i,j} \mid c \in C, 1 \leq i \leq n, \{q_i^j, \neg q_i^j\} \cap c \neq \emptyset, l = m \bmod N_i^j\}.$$

Noting that the numbers N_i^j for fixed j are co-prime, we have that the sets $P_m \cap X^j$ cycle with period $\Pi_{i=1}^n N_i^j$. More precisely, we have the following properties:

P1. For each function $f : \mathbb{A} \longrightarrow \mathbb{N}$ such that $0 \leq f(i) < N_i^j$ for each $i \in \mathbb{A}$, there exists m such that $P_m \cap X^j = \{x_{f(i)}^{c,i,j} \mid c \in C, \{q_i^j, \neg q_i^j\} \cap c \neq \emptyset\}$.

P2. If c and c' are clauses with $\{q_i^j, \neg q_i^j\} \cap c \neq \emptyset$ and $\{q_i^j, \neg q_i^j\} \cap c' \neq \emptyset$, then for all $m \in \mathbb{N}$ and $0 \leq l < N_i^j$, we have $x_l^{c,i,j} \in P_m$ iff $x_l^{c',i,j} \in P_m$.

We now label the states with propositions as follows:

1. For each $j = 1 \dots l$, there is a proposition $\texttt{level}_j$, which holds just at states of the form $x_l^{c,i,j}$ for some c, i, l.

2. For each level of quantification $j = 1 \dots k$, there is a proposition $\texttt{passgt}_j$, which we assign to be true at all states $x_l^{c,i,j}$ if $j = 1$ and at states $x_l^{c,i,j}$ with $j > 1$ iff l is divisible by $N_i^j / p_i^j = N_i^{j-1}$. Thus, there are p_i^j such states on the cycle. Intuitively, $\texttt{passgt}_j$ holds at states that represent *possible* contributions to truth assignments to the level j variables: we treat proposition q_i^j as being possibly assigned a value of *true* at a state $x_l^{c,i,j}$ satisfying $\texttt{passgt}_j$ if l is even. Note that if we consider different clauses c, c', then, for states labelled $\texttt{passgt}_j$, property **P2** implies that at a given time m, all the assignments of truth value to q_i^j according to this rule are consistent.

   However, in the formula we construct, we will be interested not directly in the truth value assigned to a variable, but in whether this assignment causes a clause in which the variable occurs to be true. For this, we further label the states $x_l^{c,i,j}$ where $\texttt{passgt}_j$ holds with the proposition $\texttt{sat}_j$, provided either l is even (so q_i^j is considered true) and q_i^j occurs positively in the clause c, or l is odd (so q_i^j is considered false) and q_i^j occurs negatively in the clause c. These are the only states in the cycle where $\texttt{sat}_j$ holds. Intuitively, this represents that the clause c is satisfied because of the choice of truth value for p_i^j.

We have said that truth of $\texttt{passgt}_j$ at a state indicates that the state represents a possible contribution to an assignment of truth values to a proposition at level j. In fact, not all such occurrences will be treated as yielding assignments, but only those at times such that all states in $P_m \cap X^j$ satisfy $\texttt{passgt}_j$. It can be seen that this is the case just when m is divisible by N_i^{j-1} for all $i = 1 \dots n$, or equivalently i (since the N_i^{j-1} are co-prime), when m is divisible by $\Pi_{i=1\dots n} N_i^{j-1}$. Intuitively, this condition represents that m is a time instant from which an assignment of truth value for *all* the level j propositions q_i^j can be read off. We may capture the satisfaction of this condition by the formula

$$\texttt{Assgt}_j = K_2(\texttt{level}_j \Rightarrow \texttt{passgt}_j)$$

which expresses that all level j states at the given time instant satisfy $\texttt{passgt}$.

Suppose we are given an assignment $\pi : \mathbb{A} \longrightarrow \{0,1\}$. Let $f : \mathbb{A} \longrightarrow \mathbb{N}$ be any function with $f(i) < N_i^j$ such that $f(i)$ is divisible by N_i^{j-1} and $f(i)$ is even iff $\pi(i) = 1$. Then by property **P1**, there exists m such that $P_m \cap X_j = \{x_{f(i)}^{c,i,j} \mid c \in C, \{p_i^j, \neg p_i^j\} \cap c \neq \emptyset\}$. Thus, the assignment to the level j variables at this instant of time is exactly π.

Moreover, all these possible asssignments occur within every interval of length $N_1^j \dots N_n^j$. In particular, between any two times $m(N_1^{j-1} \dots N_n^{j-1})$ and $(m+1)(N_1^{j-1} \dots N_n^{j-1})$, the combinations of level $j-1$ states cycle through all possibilities, so we have all possible assignments to the level $j-1$ variables represented between these successive level j assignments.

Instead of reading the value of a level j variable from the assignment at the current time, we read it from the next time that all the level j variables are assigned a value. This can be captured by the following formulas. First, define the expression $\texttt{all}_j(\varphi)$ as $\bigcirc[(\texttt{Assgt}_j \Rightarrow \varphi)\,\mathcal{U}\,(\texttt{Assgt}_{j+1})]$, which says that φ holds at all points corresponding to a j level assignment that precede the next level $j+1$ assignment. The dual of this is the expression $\texttt{some}_j(\varphi)$, defined as $\neg\texttt{all}_j(\neg\varphi)$, which says that φ holds at some point before the next level $j+1$ assignment.

Next, define $\texttt{Holds}$ as $\neg K_1 \neg \beta$, where

$$\beta = \bigvee_{j=1\dots n} \texttt{next}(\texttt{Assgt}_j, \texttt{sat}_j)$$

where $\texttt{next}(\varphi, \psi)$ is the formula $\bigcirc((\neg\varphi)\,\mathcal{U}\,(\varphi \wedge \psi))$, which says that ψ holds at the next point (after the current) where φ holds. Note that, when $\texttt{Holds}$ is evaluated at point where the state is $x_l^{c,i,j}$ the definition of observability for agent 1 implies that we check β only at points where the state is of the form $x_{l'}^{c',i',j'}$ with $c' = c$. Thus, this formula corresponds to checking that some literal in c causes c to be satisfied, according to the "current assignment" to the variables, which is determined at each level by looking at the first time in the future that corresponds to a level j assignment.

We may then translate the given formula as

$$\Phi^* = \Box(\texttt{Assgt}_k \Rightarrow \texttt{some}_{k-1}\texttt{all}_{k-2}\dots(K_2\texttt{Holds}))$$

Note that during the evaluation of this formula, because of the nesting structure for the occurrence of assignments down the levels, the successor assignment for each level j is preserved whenever an operator $\texttt{all}_{j'}$ or $\texttt{some}_{j'}$ with $j' < j$ moves the point of evaluation. Thus the successor assignments used in the evaluation of `Holds` are the same as those determined by the points of evaluation for these operators. The knowledge operator K_2 may move the point of evaluation from any point (r,n) to another point (r',n) at the same time. In particular, this operator captures quantification over all the clauses c in the given QBF formula. It follows that $E \models^{\texttt{clk}} \Phi^*$ iff Φ is true. □

A.4 An algorithm scheme

We approach the model checking problem in three stages. First, given a finite environment E and a view v, we construct an infinite environment E^v that reduces the model checking problem with respect to v in E to one of model checking E^v with respect to `obs`. Second, we introduce bisimulations between environments, which, together with the previous step, may enable the problem of model checking E with respect to v to be reduced to model checking with respect to `obs` in finite state environment E' (which is of exponential size in our applications). Finally, we combine alternating Turing machine techniques with standard Büchi automata techniques to obtain the general model checking procedure (which runs in PSPACE in our applications).

Let $E = (S, I, \to, (O_i)_{i\in\mathbb{A}}, \pi, \alpha)$ be a finite environment and let v be a view. Define E^v, the *v-environment for E*, to be $(S^v, I^v, \to^v, (O_i^v)_{i\in\mathbb{A}}, \pi^v, \alpha^v)$ where:

- $S^v = \mathit{runs}(E) \times \mathbb{N}$,
- $I^v = \mathit{runs}(E) \times \{0\}$,
- $(r,m) \to^v (r',m')$ if $r' = r$ and $m' = m+1$,
- $O_i^v(r,m) = \{(r,m)\}_i^v$,
- $\pi^v(r,m) = \pi(r(m))$, and
- $(r,m) \in \alpha^v$ iff $r(m) \in \alpha$.

The following lemma states that the observational view on this (infinite) environment coincides with the view v on the original (finite) environment. Given a run r of E, write r^v for the run of E^v defined by $r^v(n) = (r,n)$ for all $n \in \mathbb{N}$.

Lemma 7. *Let $\varphi \in \mathcal{L}_{\{\bigcirc,\mathcal{U},K_1,\dots,K_n,C\}}$ and let (r,m) be a point of E.*

Then $E,(r,m) \models^v \varphi$ iff $E^v,(r^v,m) \models^{\texttt{obs}} \varphi$.

Since every run of E^v has the form r^v for some run of E, it follows that $E \models^v \varphi$ iff $E^v \models^{\mathsf{obs}} \varphi$.

Let *fpaths*(E) be the set of all fair paths of E. For $\rho \in$ *fpaths*(E) and $m \in \mathbb{N}$ let $\rho|_m$ be the fair path with $\rho|_m(j) = \rho(m+j)$, for $j \in \mathbb{N}$.

Observe that the semantics of $E,(r,n) \models^v \varphi$ refers only to the future of the points considered in unfolding the definition. To formalise this, consider the following alternate definition of a relation $E,\rho \models^* \varphi$, defined for all $\rho \in$ *fpaths*(E), not just the initialized ones:

$E,\rho \models^* p$	if $p \in \pi(\rho(0))$, where $p \in \mathit{Prop}$,
$E,\rho \models^* \varphi_1 \wedge \varphi_2$	if $E,\rho \models^* \varphi_1$ and $E,\rho \models^* \varphi_2$,
$E,\rho \models^* \neg\varphi$	if not $E,\rho \models^* \varphi$,
$E,\rho \models^* \bigcirc\varphi$	if $E,\rho|_1 \models^* \varphi$,
$E,\rho \models^* \varphi_1 \mathcal{U} \varphi_2$	if there exists $m'' \geq 0$ such that $E,\rho|_{m''} \models^* \varphi_2$ and $E,\rho|_{m'} \models^* \varphi_1$ for all m' with $0 \leq m' < m''$.
$E,\rho \models^* K_i\varphi$	if $E,\rho' \models^* \varphi$ for all $\rho' \in$ *fpaths*(E) with $O_i(\rho'(0)) = O_i(\rho(0))$
$E,\rho \models^* C_G\varphi$	if for all sequences of states $s_0, s_1, \ldots, s_k$ such that (i) $s_0 = \rho(0)$, (ii) for all $j < k$ there exists an $i \in G$ such that $O_i(s_j) = O_i(s_{j+1})$, and (iii) for all $\rho' \in$ *fpaths*(E) with $\rho'(0) = s_k$, we have $E,\rho' \models^* \varphi$.

We write $E \models^* \varphi$ if $E,r \models^* \varphi$ for all runs r of E.

For an environment E, define a state to be reachable if it occurs in some run of E. Say that *observations in E preserve reachability* if for all states s,t of E and all agents i, if s is reachable and $O_i(s) = O_i(t)$ then t is reachable.[3]

Lemma 8. *$E,(r,m) \models^{\mathsf{obs}} \varphi$ iff $E,r|_m \models^* \varphi$ when observations in E preserve reachability.*

Proof. By induction on the construction of φ. The only non-trivial cases are those for the knowledge operators. We describe the argument for $K_i\varphi$, that for $C_G\varphi$ is similar. Suppose $E,(r,m) \models^{\mathsf{obs}} K_i\varphi$. Then for all points (r',m') of E with $O_i(r(m)) = O_i(r'(m'))$ we have $E,(r',m') \models^{\mathsf{obs}} \varphi$. We show that $E,r|_m \models^* K_i\varphi$. For, let $O_i(r(m)) = O_i(\rho(0))$, where $\rho \in$ *fpaths*(E). Since observations preserve reachability, the state $\rho(0)$ is reachable, so there exists a sequence $s0 \to s1 \to \ldots s_{m'} = \rho(0)$ with $s_0 \in I$. Let r' be the sequence $s_o \ldots s_{m'-1} \cdot \rho$. Then r is a run of E and $(r,m) \overset{\mathsf{obs}}{\sim}_i (r',m')$. Hence $E,(r,m) \models^{\mathsf{obs}} \varphi$. By the inductive hypothesis, $E,r'|_{m'} \models^* \varphi$, i.e. $E,\rho \models^* \varphi$. Hence $E,r|_m \models^* K_i\varphi$.

Conversely, suppose $E,r|_m \models^* K_i\varphi$. Let (r',m') be a point with $O_i(r'(m')) = O_i(r(m))$. Then $r'|_{m'}$ is a fair path with $O_i(r'|_{m'}(0)) = O_i(r(m))$, so $E,r'|_{m'} \models^* \varphi$, hence $E,(r',m') \models^{\mathsf{obs}} \varphi$. This shows that $E,(r,m) \models^{\mathsf{obs}} K_i\varphi$. □

[3] We remark that it is always possible to ensure this by deleting the unreachable states from E, an operation that preserves satisfaction of formulas. However, this operation is undesirable in our applications since we will deal with exponential size structures, in which observations already preserve reachability.

Next, we introduce a notion of bisimulation on environments (cf. (Park 1981)) in order to reduce the infinite state space of E^v to a finite one while preserving validity of formulas with respect to obs. For environments $E = (S, I, \rightarrow, (O_i)_{i\in\mathbb{A}}, \pi, \alpha)$ and $E' = (S', I', \rightarrow', (O'_i)_{i\in\mathbb{A}}, \pi', \alpha')$, a function $\sigma : S \longrightarrow S'$ is said to be a *bisimulation* from E to E' if the following hold:

1. $I' = \sigma(I)$,
2. if $s \rightarrow s'$ then $\sigma(s) \rightarrow' \sigma(s')$,
3. if $\sigma(s) \rightarrow' u$ then there exists $s' \in S$ such that $\sigma(s') = u$ and $s \rightarrow s'$,
4. if $O_i(s) = O_i(t)$ then $O'_i(\sigma(s)) = O'_i(\sigma(t))$,
5. if $O'_i(\sigma(s)) = O'_i(u)$ then there exists a state $t \in S$ such that $O_i(s) = O_i(t)$ and $\sigma(t) = u$.
6. $\pi' \circ \sigma = \pi$, and
7. $\sigma(s) \in \alpha'$ iff $s \in \alpha$.

Lemma 9. *Suppose that σ is a bisimulation from E to E'. Then*

1. *for all (initialised) $\rho \in fpaths(E)$, $\sigma(\rho)$ is a (initialised) fair path of E';*
2. *for all $\rho' \in fpaths(E')$ and (initial) states s of E, if $\sigma(s) = \rho'(0)$, then there exists a (initialised) $\rho \in fpaths(E)$ with $\rho(0) = s$ such that $\sigma(\rho) = \rho'$;*
3. *for all $\rho \in fpaths(E)$ we have $E, \rho \models^* \varphi$ iff $E', \sigma(\rho) \models^* \varphi$.*

Proof. Let σ be a simulation from E to E'. Part (1) follows from points 1, 2, and 7. Part (2) follows from points 3, and 7.

For part (3), let $\rho \in fpaths(E)$. We proceed by induction on the construction of φ. The propositional case is immediate from 6. The temporal cases are straightforward.

For the knowledge case, assume $E, \rho \models^* K_i\psi$ and that $O'_i(\sigma(\rho(0))) = O'_i(\rho''(0))$ for some $\rho'' \in fpaths(E)$. By 3, there exists a state t of E such that $O_i(\rho(0)) = O_i(t)$ and $\sigma(t) = \rho''(0)$. By part (2), there exists a $\rho' \in fpaths(E)$ such that $\rho'(0) = t$ and $\sigma(\rho') = \rho''$. Thus $E, \rho' \models^* \psi$. By the induction hypothesis, $E', \rho'' \models^* \psi$. This shows that $E', \sigma(\rho) \models^* K_i\psi$.

Conversely, suppose $E', \sigma(\rho) \models^* K_i\psi$. Suppose $O_i(\rho(0)) = O_i(\rho'(0))$ where $\rho' \in fpaths(E)$. By part (1), $\sigma(\rho')$ is a fair path of E'. By 4, $O'_i(\sigma(\rho(0))) = O'_i(\sigma(\rho')(0))$. Thus $E', \sigma(\rho') \models^* \psi$. By the induction hypothesis, $E', \rho' \models^* \psi$. This shows that $E, \rho \models^* K_i\psi$.

The case for common knowledge follows by similar arguments. □

Noting that all states of E^v are reachable, we obtain the following:

Corollary 10. *For all environments E and E', if there exists a bisimulation from E^v to E', then $E \models^v \varphi$ iff $E' \models^* \varphi$.*

This result provides the basic reduction that we use to obtain our complexity results. We now show that the relation $E' \models^* \varphi$ is decidable for finite environments E'. However, we will need to deal with the fact that the structure E' will be of size exponential in the size of E in our applications. For this reason, we express our decision procedure for $\models^*$ as an alternating computation (Chandra, Kozen, and Stockmeyer 1981), in which we guess and verify the components of E'.

We begin with a reduction to well-known techniques for LTL. Say that a formula is a *pure knowledge formula* if it is of the one of the forms $K_i\psi$ or $C_G\psi$, or their negation. Note that for formulas φ that are either atomic propositions or their negation, or pure knowledge formulas, we have that if $\rho(0) = \rho'(0)$, then $E, \rho \models^* \varphi$ iff $E, \rho' \models^* \varphi$. Thus, for such formulas φ, we may define $E, s \models^* \varphi$, where s is a state of E, to hold if $E, \rho \models^* \varphi$ for some (equivalently, every) path ρ with $\rho(0) = s$.

We may use this state-dependence property to transform the $\mathcal{L}_{\{\bigcirc,\mathcal{U},K_1,\ldots,K_n,C\}}$ model checking problem with respect to $\models^*$ into a problem of model checking $\mathcal{L}_{\{\bigcirc,\mathcal{U}\}}$, by replacing the pure knowledge subformulas by atomic propositions. Introduce a new atomic proposition $q_{K_i\psi}$ for each formula $K_i\psi$ and $q_{C_G\psi}$ for each formula $C_G\psi$. Let $\mathcal{L}^*_{\{\bigcirc,\mathcal{U}\}}$ be the language of temporal logic over the set of atomic propositions *Prop* together with these new atomic propositions. Given a formula φ of $\mathcal{L}_{\{\bigcirc,\mathcal{U},K_1,\ldots,K_n,C\}}$ and an occurrence of a pure knowledge formula as a subformula of φ, say this occurrence is *maximal* if it does not lie within the scope of a knowledge or common knowledge operator. For example, in $(K_2 \bigcirc K_1 p) \vee K_1 p$, the maximal occurrences of knowledge subformulas are the occurrence of $K_2 \bigcirc K_1 p$ and the second (but not the first) occurrence of $K_1 p$. Define φ^* to be the formula of $\mathcal{L}^*_{\{\bigcirc,\mathcal{U}\}}$ obtained by replacing each maximal occurrence of a knowledge formula $K_i\psi$ by the proposition $q_{K_i\psi}$ and similarly for the maximal occurrences of $C_G\psi$.

More precisely,

$$p^* = p \qquad (\varphi_1 \wedge \varphi_2)^* = {\varphi_1}^* \wedge {\varphi_2}^* \qquad (\neg\varphi)^* = \neg(\varphi^*) \qquad (\bigcirc\varphi)^* = \bigcirc(\varphi^*)$$
$$(\varphi_1 \,\mathcal{U}\, \varphi_2)^* = {\varphi_1}^* \,\mathcal{U}\, {\varphi_2}^* \qquad K_i\varphi^* = q_{K_i\psi} \qquad C_G\varphi^* = q_{C_G\psi}$$

Thus, $((K_2 \bigcirc K_1 p) \vee K_1 p)^* = q_{K_2 \bigcirc K_1 p} \vee q_{K_1 p}$. Write $Prop_{\varphi^*}$ for the set of atomic propositions occuring in φ^* and $KProp_{\varphi^*}$ for the set of atomic propositions of the form $q_{K_i\psi}$ and $q_{C_G\psi}$ that occur in φ^*.

Suppose we enrich the structure E by extending the valuation π so that $q_{K_i\psi} \in \pi(s)$ iff $E, s \models^* K_i\psi$ and $q_{C_G\psi} \in \pi(s)$ iff $E, s \models^* C_G\psi$. Call the resulting structure E^*. Then we have $E, \rho \models^* \varphi$ iff $E^*, \rho \models \varphi^*$. This turns the problem of model checking $\mathcal{L}_{\{\bigcirc,\mathcal{U},K_1,\ldots,K_n,C\}}$ in E into the problem of model checking $\mathcal{L}^*_{\{\bigcirc,\mathcal{U}\}}$ in E^*. Of course, to apply this technique, we need to have the appropriate extension E^* of E. We may deal with this in an NPSPACE computation by *guessing* the extension E^*, iteratively verifying its correctness over larger and larger pure knowledge subformulas of φ (using LTL model checking techniques), and then model checking the formula φ^*. Since NPSPACE = PSPACE, this already yields a proof of Theorem 1.[4]

[4] The guess and verify technique discussed here is essentially that used in Vardi's results on verifying implementations of knowledge-based programs (Vardi 1996).

However, in our applications, we will not be interested in a given structure E, but in a structure E' of size exponential in the size of E. This means that the cost of guessing $(E')^*$ is exponential. We will handle this by guessing the extension not upfront, but on the fly, for each state of E' as it arises during the verification, and using an APTIME computation that incorporates a Büchi automaton emptiness check for the LTL parts of the verification.

Let $\mathcal{M}_{\varphi^*}$ be the nondeterministic Büchi automaton for the $\mathcal{L}^*_{\{\bigcirc,\mathcal{U}\}}$ formula φ^* over propositions $Prop_{\varphi^*}$, with states S_{φ^*}, initial states I_{φ^*}, transitions $\Rightarrow_a$ (where $a \in \mathcal{P}(Prop_{\varphi^*})$) and acceptance condition α_{φ^*}. We make use of the following properties of this automaton (Vardi and Wolper 1984): (1) The automaton is of size $O(2^{|\varphi^*|})$, where each state is of size $O(|\varphi|)$. (2) Deciding S_{φ^*}, I_{φ^*}, $\Rightarrow_a$, and α_{φ^*} can be done in ATIME($\log_2 |\varphi|$).

For a finite environment $E = (S, I, \rightarrow, (O_i)_{i\in\mathbb{A}}, \pi, \alpha)$, we define the product $E \times \mathcal{M}_{\varphi^*}$ (a transition system with Büchi acceptance condition) as follows.

- The transition system has states $\langle b, s, v\rangle$, where $b \in \mathcal{P}(\{0,1\})$, $s \in S$ and $v \in S_{\varphi^*}$. Intuitively, $0 \in b$ ($1 \in b$) represents that E (resp., $\mathcal{M}_{\varphi^*}$), has passed through an accepting state since the most recent accepting state of the product.
- The set of initial states consists of all $\langle \emptyset, s, v\rangle$ where $s \in I$ and $v \in I_{\varphi^*}$.
- There is a transition $\langle b, s, v\rangle \Rightarrow_k \langle b', s', v'\rangle$ for a set $k \subseteq KProp_{\varphi^*}$ when:
  - $s \rightarrow s'$,
  - $v \Rightarrow_{\pi(s)\cup k} v'$, and
  - $b' = b_0 \cup b_1 \cup b_2$, where if $b = \{0,1\}$ then $b_0 = \emptyset$, else $b_0 = b$; if $s \in \alpha$ then $b_1 = \{0\}$, else $b_1 = \emptyset$; and if $v \in \alpha_{\varphi^*}$ then $b_2 = \{1\}$, else $b_2 = \emptyset$;
- the automaton has as accepting states the states $\langle b, s, v\rangle$ with $b = \{0,1\}$.

Intuitively, this transition system represents running $\mathcal{M}_{\varphi^*}$ as a monitor on runs of E, with the values of the propositions $KProp_{\varphi^*}$ chosen arbitrarily. Thus, there exists a fair path $\rho = s_0 s_1 \dots$ of E such that $E, \rho \models^* \varphi$ iff there exists an accepting run $\langle b_0, s_0, v_0\rangle \Rightarrow_{k_0} \langle b_1, s_1, v_1\rangle \Rightarrow_{k_1} \langle b_2, s_2, v_2\rangle \Rightarrow_{k_2} \dots$ of $E \times \mathcal{M}_{\neg\varphi^*}$ such that for all $j \geq 0$, we have $E, s_j \models^* k_j$. Applying the usual emptiness check for Büchi automata, such a path exists iff we can find a finite such sequence with $\langle b_l, s_l, v_l\rangle$ an accepting state and final element $\langle b_{l'}, s_{l'}, v_{l'}\rangle = \langle b_l, s_l, v_l\rangle$ for some $l' > l$, where both l and $l' - l$ are at most $|E \times \mathcal{M}_{\varphi^*}|$. Our decision procedure searches for such paths using a Savitch-style reachability procedure (Savitch 1970) in order to deal with the exponential size of the search-space.

For the verification that $E, s \models^* k$, it suffices to check, for each maximal knowledge subformula $K_i\psi$ of φ, that $q_{K_i\psi} \in k$ iff $O_i(s) = O_i(t)$ implies that for all fair paths $\rho = t_0 t_1 \dots$ with $t_0 = t$, we have $E, \rho \models \psi^*$. For this, we recursively apply the above ideas on $E \times \mathcal{M}_{\neg\psi^*}$. Since ψ is a strict subformula of φ, the recursion is well founded. A similar check is applied for the common knowledge subformulas.

We are now ready to present our general algorithm scheme as an alternating computation (Chandra et al. 1981). Suppose that we are given a finite environment E, for which it is known that there exists a bisimulation from E^v to a finite environment $E' = (S', I', \rightarrow', (O'_i)_{i \in \mathbb{A}}, \pi', \alpha')$. We assume that there is a representation of E' such that the states and other components of E' can be represented and verified within known space and alternating time complexity bounds. (That is, given E, the states of E' are representable as strings of length bounded by some known function of $|E|$, in such a way that we can decide whether such a string represents a state of E', whether $s \rightarrow' s'$ etc. with some known complexity bounds.) We define the following alternating procedure that searches for such runs by operating over the states $\langle b, s, v\rangle$ of the automata $E' \times \mathcal{M}_{\psi^*}$ for subformulas ψ of φ and their negations. For clarity, we write expressions referring to the components of E' (such as "choose $s \in I'$ and do X") which need to be expanded to expressions ("choose s and universally (1) verify $s \in I'$ and (2) do X") that use the verification routines assumed to exist.

$\mathtt{VERIFY}(E, \varphi)$**:** Universally choose $s \in I'$ and call $\neg\mathtt{FALSIFY}(E, s, \varphi)$

$\mathtt{FALSIFY}(E, s, \psi)$**:** Existentially choose $k \subseteq KProp_{\psi^*}$, an initial state v of $\mathcal{M}_{\neg\psi^*}$, an accepting state $\langle b_0, s_0, v_0\rangle$ and a state $\langle b_1, s_1, v_1\rangle$ of $E' \times \mathcal{M}_{\neg\psi^*}$ where $(b_0, s_0, v_0) \Rightarrow_k (b_1, s_1, v_1)$.
Let $N = \lceil log_2 \left|states(E' \times \mathcal{M}_{\neg\psi^*})\right|\rceil$.
Universally call:

- $\mathtt{REACH}(E, (\emptyset, s, v), (b_0, s_0, v_0), N, \neg\psi)$,
- $\mathtt{CHECK}(E, s_0, k, \psi)$, and
- $\mathtt{REACH}(E, (b_1, s_1, v_1), (b_0, s_0, v_0), N, \neg\psi)$

$\mathtt{CHECK}(E, s, k, \psi)$**:** Universally,

- for each $p_{K_i\psi'}$ in $KProp_{\psi^*}$,
  if $p_{K_i\psi'} \in k$ then call $\mathtt{KCHECK}(E, s, K_i\psi')$ else call $\neg\mathtt{KCHECK}(E, s, K_i\psi')$
- for each $p_{C_G\psi'}$ in $KProp_{\psi^*}$,
  if $p_{C_G\psi'} \in k$ then call $\mathtt{CKCHECK}(E, s, C_G\psi')$ else call $\neg\mathtt{CKCHECK}(E, s, C_G\psi')$

$\mathtt{KCHECK}(E, s, K_i\psi)$**:** Universally, for each $s' \in S'$ where $O'_i(s) = O'_i(s')$, call $\neg\mathtt{FALSIFY}(E, s', \psi)$

$\mathtt{CKCHECK}(E, s, C_G\psi)$**:** Universally, for each $s' \in S'$: (1) verify[5] there is a sequence $s = s_0, \ldots, s_k = s'$ with $k \leq |S'|$ and for each $j < k$ there is an $i \in G$ such that $O'_i(s_j) = O'_i(s_{j+1})$. (2) call $\neg\mathtt{FALSIFY}(E, s', \psi)$

$\mathtt{REACH}(E, (b_0, s_0, v_0), (b_1, s_1, v_1), N, \psi)$**:** Accept if $(b_0, s_0, v_0) = (b_1, s_1, v_1)$.
Otherwise if $N = 0$, existentially guess $k \subseteq KProp_{\psi^*}$ then

- universally verify that $(b_0, s_0, v_0) \Rightarrow_k (b_1, s_1, v_1)$ and $\mathtt{CHECK}(E, s_0, k, \psi)$.

[5] In general, this may require another Savitch-style search. In fact, in our applications, $k \leq |S|^2$, i.e., the square of the number of states of E, will suffice, so this is not necessary.

If $N > 0$, existentially guess a state (b_2, s_2, v_2) of $E \times \mathcal{M}_{\psi^*}$, then universally call:

- $\texttt{REACH}(E, (b_0, s_0, v_0), (b_2, s_2, v_2), N-1, \psi)$ and
- $\texttt{REACH}(E, (b_2, s_2, v_2), (b_1, s_1, v_1), N-1, \psi)$.

An analysis of the complexity of the algorithm scheme yields the following.

Theorem 11. *Let v be a view, and $\mathscr{C}$ be a class of environments such that for each environment $E \in \mathscr{C}$ there exists an environment E' with states that can be represented in space $f(|E|)$ and components that can be verified in ATIME$(g(|E|))$, such that there is a bisimulation σ from E^v to E'. Then $\{ (E,\varphi) \in \mathscr{C} \times \mathscr{L}_{\{\bigcirc,\mathscr{U},K_1,\dots,K_n,C\}} \mid E \models^v \varphi \}$ is in ATIME$(p(f(|E|), g(|E|), |\varphi|))$ for some polynomial p.*

Proof. Correctness of the alternating procedure is a straightforward combination of the correctness arguments for Büchi automaton emptiness checking, Savitch-style search and the definition of $\models^*$.

For the complexity analysis, note that the number N used in $\texttt{FALSIFY}(E, s, \psi)$ is $O(f(|E|) + |\psi|)$. The routine $\texttt{FALSIFY}(E, s, \psi)$ generates a computation tree in which the longest branch is $O(f(|E|) + |\psi|)$ (for the existential choice) plus the maximum of $O(g(|E|))$ (for the verification of the guessed components) and the longest branch for $\texttt{REACH}(E, w, w', N, \psi)$.

Note $\texttt{REACH}(E, w, w', n, \psi)$ calls $\texttt{CHECK}()$ only when $n = 0$, and each recursion before then adds time $O(f(|E|) + |\psi|)$ to construct the guess for the recursive call. Hence $\texttt{REACH}(E, w, w', N, \psi)$ runs in alternating time $O((f(|E|) + |\psi|)^2)$ plus the time required for the call to $\texttt{CHECK}(E, s, k, \psi)$ once $n = 0$. The largest cost in the latter is the calls to $\texttt{CKCHECK}(E, s, C_G\psi')$, which add another $O((f(|E|) + |\psi|)^2)$ alternating time steps before calling $\texttt{FALSIFY}(E, s, \psi')$, with ψ' of lower knowledge depth than ψ. Thus, if $T(E, h)$ is the alternating time required by $\texttt{FALSIFY}(E, s, \psi)$ for formulas ψ with $|\psi| \leq h$, we have the recurrence $T(E, h) = O((f(|E|) + h)^2 + g(|E|)) + T(E, h-1)$, hence $T(E, h) = O(h \cdot ((f(|E|) + h)^2 + g(|E|)))$. This yields the result. □

We remark that since the procedure REACH has an alternation before the recursive call, the number of alternations is also polynomial in $|E|$. Theorem 5 can be understood as asserting that this is inherently so.

In the following sections, we apply Theorem 11 to obtain complexity bounds for model checking the logic of knowledge and linear time in a number of cases. In each case, we identify an appropriate environment E' where the states can be represented and verified in polynomial space and time, respectively, hence the complexity of the alternating procedure is APTIME. By (Chandra et al. 1981), this is equivalent to PSPACE. The environments E' and the bisimulations we use are extensions (by the addition of transition relations $\rightarrow'$) of similar structures that have been used elsewhere in the literature (van der Meyden 1996b) for another problem (existence of finite-state implementations of knowledge-based programs.)

A.5 Model checking with respect to perfect recall

In this section we consider several special cases of model checking with respect to perfect recall. The first restricts formulas to refer only to the knowledge of a single agent, and the latter concerns model checking the full language $\mathscr{L}_{\{\bigcirc,\mathscr{U},K_1,\ldots,K_n,C\}}$ in restricted environments.

A.5.1 Formulas of $\mathscr{L}_{\{\bigcirc,\mathscr{U},K\}}$

We first treat the case of model checking formulas of a single agent (agent 1) using the perfect recall view. (This case may also be applied to model checking formulas that refer only to a single agent's knowledge, simply by dropping the other agents' observation functions from the environment.)

In this setting it suffices to track the set of states the agent considers possible at each point in time. We define the environment $E' = (S', I', \rightarrow', O'_1, \pi', \alpha')$ by:

- $S' = \{ (s, P) \mid s \in S, P \subseteq S, s \in P \}$
- $I' = \{ (s, P_0(s)) \mid s \in I \}$ where $P_0(s) = \left\{ s' \in I \;\middle|\; O_1(s) = O_1(s') \right\}$
- $(s, P) \rightarrow' (s', P')$ iff $s \rightarrow s'$ and $P' = \left\{ t' \;\middle|\; t \in P, t \rightarrow t', O_1(t') = O_1(s') \right\}$, and
- $O'_1(s, P) = P$.

with bisimulation from E^{pr} given by $\sigma(r, m) = (r(m), P_1^{\mathrm{pr}}(E, r, m))$. Observe that a state of E' can be represented in space $O(\log_2 |S| + |S|)$. States of E' can be seen to be a special case (1-trees) of data structures previously used in (van der Meyden 1998) for model checking $\mathscr{L}_{\{K_1,\ldots,K_n\}}$. That σ is a bisimulation can be seen by arguments in that work. It is easy to check that observations preserve reachability. By Theorem 11 we conclude that this model checking problem can be decided in PSPACE, which completes the proof of Theorem 2.

A.5.2 Multi-agent broadcast and $\mathscr{L}_{\{\bigcirc,\mathscr{U},K_1,\ldots,K_n,C\}}$ with perfect recall

Broadcast environments (van der Meyden 1996b; van der Meyden and Wilke 2005) model situations in which agents may maintain private information, but where the only means by which this information can be communicated is by synchronous simultaneous broadcast to all agents.

We give a definition of broadcast environments here that is slightly more abstract than previous formulations, which dealt with a notion of environment in which agents are equipped with actions that they may perform. Formally, we define a broadcast environment to be an environment $E = (S, I, \rightarrow, (O_i)_{i\in\mathbb{A}}, \pi, \alpha)$ in which the states and observation functions and transition relation have a particular structure.

- The set S of states of E is a subset of $S_0 \times S_1 \times \ldots \times S_n$, where S_0 is a finite set of *shared states*, and for each agent i a set S_i of *private states*. If $s = \langle s_0, \ldots, s_n\rangle$ denotes a state, we write $\mathbf{p}_i(s)$ to denote agent i's private state s_i. For each agent i, define the binary function $\bowtie_i$ on S by $\langle s_0, s_1, \ldots s_n\rangle \bowtie_i \langle t_0, \ldots t_n\rangle = \langle s_0, s_1 \ldots s_{i-1}, t_i, s_{i+1}, \ldots s_n\rangle$. We require that S is closed under the functions $\bowtie_i$.

- The observation functions are given by $O_i(s) = (O_c(s_0), \mathbf{p}_i(s))$, where $O_c : S_0 \longrightarrow \mathcal{O}$ is a *common observation function.*

- The transition relation has the property that for each agent $i = 1 \ldots n$, if $O_i(s) = O_i(t)$ and $s \to s'$ and $t \to t'$ and $O_c(s') = O_c(t')$, then $s \to s' \bowtie_i t'$.

There is no constraint on the set of initial states. Intuitively, the common observation function models the information that is being broadcast, and the private states model private information that is being maintained by the agents. An agent's observation consists of the broadcast information and its private information. The condition on the transition relation can be understood as saying that an agent's choice of update on its private state (1) may depend only on the current observation and the incoming common observation and (2) does not affect the update on the common state or any of the other agents' updates.

Paradigmatic examples of broadcast systems are card games such as bridge (where both bidding and playing of cards can be viewed as a broadcast) and systems composed of processes attached to bus, with all processes receiving every communication (as in "snoopy cache coherence protocols" (Sweazey and Smith 1986)).

Note that every single agent system E is isomorphic to a broadcast system E'. For, if we represent a state s of E by a state $\langle s, O_1(s)\rangle$ of E', and view the first component as being the shared state and the second component as the private state of the single agent, and take $O_c(s) = O_1(s)$, then the constraint on the transition relation is trivially satisfied, because if $O_1(s) = O_1(t)$ then $(s, O_1(s)) \bowtie (t, O_1(t)) = (s, O_1(s))$.

For the broadcast, perfect recall case, we use the following environment E':

- S' is the set of elements $(s, f, t) \in I \times (I \longrightarrow \mathcal{P}(S)) \times S$ such that $t \in f(s)$,

- $(s, f, t) \in I'$ iff $s \in I$, f is given by $f(s') = \left\{ t \in I \;\middle|\; O_c(t) = O_c(s') \right\}$ for all $s' \in I$, and $t \in f(s)$,

- $(s, f, t) \to' (s, f', t')$ if $f'(s') = \left\{ u' \;\middle|\; u \in f(s'), u \to u', O_c(u') = O_c(t') \right\}$ and $t \to t'$,

- $O'_i(s, f, t) = (O_i(s), f, O_i(t))$,

and the bisimulation σ given by $\sigma(r, m) = (r(0), f, r(m))$, where $f(s)$ is the set of states t such that there exists a trace $r'[0 \ldots m]$ of E with $O_c(r'[0..m]) = O_c(r[0..m])$ and $t = r'(m)$. Observations preserve reachability in this environment. The states of E' can be represented in size $O(\log_2 |S| + |I| \cdot |S| + \log_2 |S|) = O(|S|^2)$, and applying Theorem 11 shows once again that this model checking problem is in PSPACE, completing the proof of Theorem 3. Note also that in this structure, if $(s, f, t) \overset{\text{obs}}{\sim}_G (s', f', t')$, then it does so by means of a sequence of states all of which have second component f, and also $f' = f$. Thus a maximal path length of $|S|^2$ suffices in `CKCHECK`.

A.6 Formulas of $\mathscr{L}_{\{\bigcirc,\mathscr{U},K_1,\ldots,K_n,C\}}$ for the clock and observational views

In order to model check formulas with respect to the clock view, the image of a point (r,m) in the simulating environment E' needs to keep track of the set of states that are reachable in exactly m steps. We define E' by

- $S' = \{\, (s,P) \mid s \in S, P \subseteq S, s \in P \,\}$
- $I' = I \times \{I\}$,
- $(s,P) \to' (s',P')$ if $s \to s'$ and $P' = \{t' \mid t \in P, t \to t'\}$.
- $O'_i(s,P) = (O_i(s),P)$

The bisimulation is given by $\sigma(r,m) = (r(m), \{\, r'(m) \mid r' \in runs(E) \,\})$. Observations can be seen to preserve reachability. The states in the constructed environment can be represented in space $O(\log|S|+|S|)$. This problem is again in PSPACE by Theorem 11. This yields a proof of Theorem 4. In this structure, if $(s,P) \overset{\text{obs}}{\sim}_G (s',P')$, then it does so by means of a sequence of states all of which have second component P, and also $P' = P$. Thus a maximal path length of $|S|$ suffices in CKCHECK.

We can already decide realization in a finite environment E with respect to the observational semantics by furnishing a standard LTL model checker with the equivalence classes induced by the observation function. To remain within our framework, it suffices to use an environment identical to E with bisimulation $\sigma(r,m) = r(m)$ from E^{obs} to E. Its states can be represented in size $O(\log|S|)$. However, in order for observations to preserve reachability in this case, we need to first remove unreachable states from the environment. Here also a maximal path length of $|S|$ suffices in CKCHECK.

A.7 Conclusion

We have shown that our general bisimulation-based scheme for model checking the logic of knowledge and linear time yields PSPACE complexity bounds in a number of interesting cases of the general problem (which has much higher complexity).

Our notion of bisimulation allows reductions on the temporal structure of environments, but we have not exploited this in our applications. It could be worth exploring this observation in practice. Experiments conducted by Fisler and Vardi (Fisler and Vardi 1999) suggest that bisimulation reduction is of limited utility for temporal logic model checking, but arguments of van der Meyden and Zhang (van der Meyden and Zhang 2007) suggest such reductions might be effective for the much larger search spaces produced when dealing with information flow properties.

The techniques are also applicable to show decidability for certain other classes of environments (with higher complexity bounds). We leave the details for elsewhere. We believe that the techniques we have developed can also be adapted to deal with the combination of branching time and the logic of knowledge: we leave this for future work.

Wozna et al have studied model checking a logic of knowledge and branching time in a real time systems modelled using timed automata (Wozna, Lomuscio, and Penczek 2005). Their semantics is close to our clock semantics, but we note that their until operator is bounded to a specific interval, so the closest appropriate comparison is to our language $\mathcal{L}_{\{\bigcirc,K_1,\ldots,K_n,C\}}$. They give decidability but not complexity results, but study bounded model checking techniques for their logic.

Appendix B

The Worker/Wrapper Transformation

THE worker/wrapper transformation has been formalised by Gill and Hutton (2009) as a technique for changing "a computation of one type into a worker of a different type, together with a wrapper that acts as an impedance matcher between the original and new computations." It is intended to be used in the optimising passes of compilers (Peyton Jones and Launchbury 1991), and also for high-level proofs in the style of the *calculating compilers* work of Meijer (1992), and the broader Squiggol enterprise (Meijer et al. 1991). It has been used by Gill and Farmer (2011) to "semi-formally" refine circuits described in a Lava style (§4.2.4) to hardware.

This appendix describes a mechanisation of the results of Gill and Hutton (2009). We also describe a correct fusion rule and prove its correctness in §B.3, and provide a new example in §B.4. This work was reported in Gammie (2009) and Gammie (2011c).

Here we use Isabelle/HOLCF, due to Müller, Nipkow, von Oheimb, and Slotosch (1999) and more recently Huffman (2012), which provides mechanical support for reasoning about denotational semantics through an embedding of Scott's LCF logic in HOL. In particular, Λ _._ denotes continuous function abstraction, _ $\cdot$ _ continuous function application and _ oo _ continuous function composition. The **domain** command defines recursive datatypes. The other notation is close to mathematical practice.

B.1 Fixed-point theorems for program transformation

We begin with a pair of theorems from the early days of denotational semantics. The origins of these results are lost to history; the interested reader can find some of it in Bekić (1984); de Bakker, de Bruin, and Zucker (1980); Greibach (1975); Harel (1980); Manna (1974); Plotkin (1983); Sangiorgi (2009); Stoy (1977); Winskel (1993).

The *rolling rule* captures what intuitively happens when we re-order a recursive computation consisting of two parts. This theorem dates from the 1970s at the latest – see Stoy (1977, p210) and Plotkin (1983). The following proofs were provided by Gill and Hutton (2009).

```
lemma rolling_rule_ltr: "fix·(g oo f) ⊑ g·(fix·(f oo g))"
proof -
  have "g·(fix·(f oo g)) ⊑ g·(fix·(f oo g))"
    by (rule below_refl) — reflexivity
  hence "g·((f oo g)·(fix·(f oo g))) ⊑ g·(fix·(f oo g))"
    using fix_eq[where F="f oo g"] by simp — computation
  hence "(g oo f)·(g·(fix·(f oo g))) ⊑ g·(fix·(f oo g))"
    by simp — re-associate op oo
  thus "fix·(g oo f) ⊑ g·(fix·(f oo g))"
    using fix_least_below by blast — induction
qed

lemma rolling_rule_rtl: "g·(fix·(f oo g)) ⊑ fix·(g oo f)"
proof -
  have "fix·(f oo g) ⊑ f·(fix·(g oo f))" by (rule rolling_rule_ltr)
  hence "g·(fix·(f oo g)) ⊑ g·(f·(fix·(g oo f)))"
    by (rule monofun_cfun_arg) — g is monotonic
  thus "g·(fix·(f oo g)) ⊑ fix·(g oo f)"
    using fix_eq[where F="g oo f"] by simp — computation
qed

lemma rolling_rule: "fix·(g oo f) = g·(fix·(f oo g))"
  by (rule below_antisym[OF rolling_rule_ltr rolling_rule_rtl])
```

Least-fixed-point fusion provides a kind of induction that has proven to be very useful in calculational settings. Intuitively it lifts the step-by-step correspondence between `f` and `h` witnessed by the strict function `g` to the fixed points of `f` and `g`:

$$
\begin{tikzcd}
\bot & & \bullet \arrow[r, "h"] & \bullet & & \mathrm{fix}\ h \\
\bot \arrow[u, "g"] & \wedge & \bullet \arrow[u, "g"] \arrow[r, "f"'] & \bullet \arrow[u, "g"'] & \Longrightarrow & \mathrm{fix}\ f \arrow[u, "g"']
\end{tikzcd}
$$

Fokkinga and Meijer (1991), and also their later collaboration with Paterson [1991], made extensive use of this rule, as did Tullsen (2002) in his program transformation tool PATH. This diagram is strongly reminiscent of the simulations used to establish refinement relations between imperative programs and their specifications (de Roever and Engelhardt 1998).

The following proof is close to the third variant of Stoy (1977, p215). We relate the two fixpoints using the rule `parallel_fix_ind`:

$$
\frac{\mathtt{adm}\ (\lambda x.\ \mathtt{P}\ (\mathrm{fst}\ x)\ (\mathrm{snd}\ x)) \qquad \mathtt{P}\ \bot\ \bot \qquad \bigwedge x\ y.\ \dfrac{\mathtt{P}\ x\ y}{\mathtt{P}\ (\mathtt{F}\cdot x)\ (\mathtt{G}\cdot y)}}{\mathtt{P}\ (\mathrm{fix}\cdot\mathtt{F})\ (\mathrm{fix}\cdot\mathtt{G})}
$$

in a very straightforward way:

For a recursive definition $comp = \mathsf{fix}\ body$ for some $body :: A \to A$ and a pair of functions $wrap :: B \to A$ and $unwrap :: A \to B$ where $wrap \circ unwrap = id_A$, we have:

$$
\begin{aligned}
&comp = wrap\ work \\
&\quad work :: B \\
&\quad work = \mathsf{fix}\ (unwrap \circ body \circ wrap)
\end{aligned}
\qquad \text{(the worker/wrapper transformation)}
$$

Also:

$$(unwrap \circ wrap)\ work = work \qquad \text{(worker/wrapper fusion)}$$

Figure B.1: The worker/wrapper transformation and fusion rule of Gill and Hutton (2009).

```
lemma lfp_fusion:
  assumes "g·⊥ = ⊥"
  assumes "g oo f = h oo g"
  shows "g·(fix·f) = fix·h"
proof(induct rule: parallel_fix_ind)
  case 2 show "g·⊥ = ⊥" by fact
  case (3 x y) from ‘g·x = y‘ ‘g oo f = h oo g‘ show "g·(f·x) = h·y"
    by (simp add: cfun_eq_iff)
qed simp
```

This lemma also goes by the name of *Plotkin's axiom* (Pitts 1996) or *uniformity* (Simpson and Plotkin 2000).

B.2 The transformation according to Gill and Hutton

The worker/wrapper transformation and associated fusion rule as formalised by Gill and Hutton (2009) are reproduced in Figure B.1, and the reader is referred to the original paper for further motivation and background.

Armed with the rolling rule, Gill and Hutton show that the worker/wrapper transformation is sound. There is a battery of these transformations with varying strengths of hypothesis.

The first requires `wrap oo unwrap` to be the identity for all values.

```
lemma worker_wrapper_id:
  fixes wrap :: "'b::pcpo → 'a::pcpo"
  fixes unwrap :: "'a → 'b"
  assumes wrap_unwrap: "wrap oo unwrap = ID"
  assumes comp_body: "computation = fix·body"
  shows "computation = wrap·(fix·(unwrap oo body oo wrap))"
proof -
  from comp_body have "computation = fix·(ID oo body)"
    by simp
```

```
  also from wrap_unwrap have "... = fix·(wrap oo unwrap oo body)"
    by (simp add: assoc_oo)
  also have "... = wrap·(fix·(unwrap oo body oo wrap))"
    using rolling_rule[where f="unwrap oo body" and g="wrap"]
    by (simp add: assoc_oo)
  finally show ?thesis .
qed
```

The second weakens this assumption by requiring that `wrap oo wrap` only act as the identity on values in the image of `body`.

```
lemma worker_wrapper_body:
  fixes wrap :: "'b::pcpo → 'a::pcpo"
  fixes unwrap :: "'a → 'b"
  assumes wrap_unwrap: "wrap oo unwrap oo body = body"
  assumes comp_body: "computation = fix·body"
  shows "computation = wrap·(fix·(unwrap oo body oo wrap))"
proof -
  from comp_body have "computation = fix·(wrap oo unwrap oo body)"
    using wrap_unwrap by (simp add: assoc_oo wrap_unwrap)
  also have "... = wrap·(fix·(unwrap oo body oo wrap))"
    using rolling_rule[where f="unwrap oo body" and g="wrap"]
    by (simp add: assoc_oo)
  finally show ?thesis .
qed
```

This is particularly useful when the computation being transformed is strict in its argument.

Finally we can allow the identity to take the full recursive context into account. This rule was described by Gill and Hutton but not used.

```
lemma worker_wrapper_fix:
  fixes wrap :: "'b::pcpo → 'a::pcpo"
  fixes unwrap :: "'a → 'b"
  assumes wrap_unwrap: "fix·(wrap oo unwrap oo body) = fix·body"
  assumes comp_body: "computation = fix·body"
  shows "computation = wrap·(fix·(unwrap oo body oo wrap))"
proof -
  from comp_body have "computation = fix·(wrap oo unwrap oo body)"
    using wrap_unwrap by (simp add: assoc_oo wrap_unwrap)
  also have "... = wrap·(fix·(unwrap oo body oo wrap))"
    using rolling_rule[where f="unwrap oo body" and g="wrap"]
    by (simp add: assoc_oo)
  finally show ?thesis .
qed
```

Gill and Hutton's `worker_wrapper_fusion` rule is intended to allow the transformation of `(unwrap oo wrap)·R` to `R` in recursive contexts, where `R` is meant to be a self-call. Note that it assumes that the first worker/wrapper hypothesis can be established.

```
lemma worker_wrapper_fusion:
  fixes wrap :: "'b::pcpo → 'a::pcpo"
  fixes unwrap :: "'a → 'b"
  assumes wrap_unwrap: "wrap oo unwrap = ID"
  assumes work: "work = fix·(unwrap oo body oo wrap)"
  shows "(unwrap oo wrap)·work = work"
proof -
  have "(unwrap oo wrap)·work = (unwrap oo wrap)·(fix·(unwrap oo body oo wrap))"
    using work by simp
  also have "...
          = (unwrap oo wrap)·(fix·(unwrap oo body oo wrap oo unwrap oo wrap))"
    using wrap_unwrap by (simp add: assoc_oo)
  also
  from rolling_rule[where f="unwrap oo body oo wrap" and g="unwrap oo wrap"]
  have "... = fix·(unwrap oo wrap oo unwrap oo body oo wrap)"
    by (simp add: assoc_oo)
  also have "... = fix·(unwrap oo body oo wrap)"
    using wrap_unwrap by (simp add: assoc_oo)
  finally show ?thesis using work by simp
qed
```

The following sections show that this rule only preserves partial correctness. This is because Gill and Hutton apply it in the context of the fold/unfold program transformation framework of Burstall and Darlington (1977), which need not preserve termination. We show that a totally correct fusion rule does require extra conditions and propose one such sufficient condition.

B.2.1 Worker/wrapper fusion is partially correct

We now examine how Gill and Hutton apply their worker/wrapper fusion rule in the context of the fold/unfold framework.

The key step of those left implicit in the original paper is the use of the fold rule to justify replacing the worker with the fused version. Schematically, the fold/unfold framework maintains a history of all definitions that have appeared during transformation, and the fold rule treats this as a set of rewrite rules oriented right-to-left. (The unfold rule treats the current working set of definitions as rewrite rules oriented left-to-right.) Hence as each definition $f = body$ yields a rule of the form $body \Longrightarrow f$, one can always derive $f = f$. Clearly this has dire implications for the preservation of termination behaviour.

Tullsen (2002) in his §3.1.2 observes that the essence of the fold rule is Park induction:

$$\frac{\mathtt{f \cdot x = x}}{\mathsf{fix}\cdot\mathtt{f} \sqsubseteq \mathtt{x}}\ \mathtt{fix\_least}$$

In general a fixed point x of f need only be an upper bound on the least fixed point $\mathit{fix}\, f$; in other words $f\ x = x$ implies only the partially correct $\mathit{fix}\, f \sqsubseteq x$ and not the totally correct $\mathit{fix}\, f = x$. We use this characterisation of the fold rule to show that if *unwrap* is non-strict (i.e. $\mathit{unwrap}\, \bot \neq \bot$) then there are programs where worker/wrapper fusion as used by Gill and Hutton need only be partially correct.

Consider the scenario described in Figure B.1. After applying the worker/wrapper transformation, we attempt to apply fusion by finding a residual expression `body'` such that the body of the worker, i.e. the expression `unwrap oo body oo wrap`, can be rewritten as `body' oo unwrap oo wrap`. Intuitively this is the semantic form of workers where all self-calls are fusible. Our goal is to justify redefining `work` to `fix·body'`, i.e. to establish:

```
fix·(unwrap oo body oo wrap) = fix·body'
```

We show that worker/wrapper fusion as proposed by Gill and Hutton is partially correct using Park induction:

```
lemma fusion_partially_correct:
  assumes wrap_unwrap: "wrap oo unwrap = ID"
  assumes work: "work = fix·(unwrap oo body oo wrap)"
  assumes body': "unwrap oo body oo wrap = body' oo unwrap oo wrap"
  shows "fix·body' ⊑ work"
proof(rule fix_least)
  have "work = (unwrap oo body oo wrap)·work"
    using work by (simp add: fix_eq[symmetric])
  also have "... = (body' oo unwrap oo wrap)·work"
    using body' by simp
  also have "... = (body' oo unwrap oo wrap)·((unwrap oo body oo wrap)·work)"
    using work by (simp add: fix_eq[symmetric])
  also have "... = (body' oo unwrap oo wrap oo unwrap oo body oo wrap)·work"
    by simp
  also have "... = (body' oo unwrap oo body oo wrap)·work"
    using wrap_unwrap by (simp add: assoc_oo)
  also have "... = body'·work"
    using work by (simp add: fix_eq[symmetric])
  finally show "body'·work = work" by simp
qed
```

The next section shows the converse does not obtain.

B.2.2 A non-strict `unwrap` may go awry

If `unwrap` is non-strict, then the fusion rule proposed by Gill and Hutton need not preserve termination. To show this we take a small artificial example. The type `A` is not important, but we need access to a non-bottom inhabitant. The target type `B` is the non-strict lift of `A`.

```
domain A = A
domain B = B (lazy "A")
```

The functions `wrap` and `unwrap` that map between these types are routine. Note that `wrap` is (necessarily) strict due to the property $\forall$`x. f·(g·x) = x` $\Longrightarrow$ `f·⊥ = ⊥`.

```
fixrec wrap :: "B → A" where "wrap·(B·a) = a"
fixrec unwrap :: "A → B" where "unwrap = B"
```

Discharging the worker/wrapper hypothesis is similarly routine.

```
lemma wrap_unwrap: "wrap oo unwrap = ID" by (simp add: cfun_eq_iff)
```

The candidate computation we transform can be any that uses the recursion parameter `r` non-strictly. The following is especially trivial.

```
fixrec body :: "A → A" where "body·r = A"
```

The transformed worker can be strict in the recursion parameter `r`, as `unwrap` always lifts it.

```
fixrec body' :: "B → B" where "body'·(B·a) = B·A"
```

As explained above, we set up the fusion opportunity:

```
lemma body_body': "unwrap oo body oo wrap = body' oo unwrap oo wrap"
  by (simp add: cfun_eq_iff)
```

This result depends crucially on `unwrap` being non-strict.

Our earlier result shows that the proposed transformation is partially correct:

```
lemma "fix·body' ⊑ fix·(unwrap oo body oo wrap)"
  by (rule fusion_partially_correct[OF wrap_unwrap refl body_body'])
```

However it is easy to see that it is not totally correct:

```
lemma "¬ fix·(unwrap oo body oo wrap) ⊑ fix·body'"
proof -
  have l: "fix·(unwrap oo body oo wrap) = B·A" by (subst fix_eq) simp
  have r: "fix·body' = ⊥" by (simp add: fix_strict)
  from l r show ?thesis by simp
qed
```

This trick works whenever `unwrap` is not strict. In the following section we show that requiring `unwrap` to be strict leads to a straightforward proof of total correctness.

Note that if we have already established that `wrap oo unwrap = ID`, then making `unwrap` strict preserves this equation:

```
lemma
  assumes "wrap oo unwrap = ID"
  shows "wrap oo strictify·unwrap = ID"
```

```
proof(rule cfun_eqI)
  fix x from assms show "(wrap oo strictify·unwrap)·x = ID·x"
    by (cases "x = ⊥") (simp_all add: cfun_eq_iff retraction_strict)
qed
```

From this we conclude that the worker/wrapper transformation itself cannot exploit any laziness in `unwrap` under the context-insensitive assumptions of `worker_wrapper_id`. This is not to say that other program transformations may not be able to.

B.3 A totally-correct fusion rule

We now show that a termination-preserving worker/wrapper fusion rule can be obtained by requiring `unwrap` to be strict. (As we observed earlier, `wrap` must always be strict due to the assumption that `wrap oo unwrap = ID`.)

Our first result shows that a combined worker/wrapper transformation and fusion rule is sound, using the assumptions of `worker_wrapper_id` and the ubiquitous `lfp_fusion` rule.

```
lemma worker_wrapper_fusion_new:
  fixes wrap :: "'b::pcpo → 'a::pcpo"
  fixes unwrap :: "'a → 'b"
  fixes body' :: "'b → 'b"
  assumes wrap_unwrap: "wrap oo unwrap = (ID :: 'a → 'a)"
  assumes unwrap_strict: "unwrap·⊥ = ⊥"
  assumes body_body': "unwrap oo body oo wrap = body' oo (unwrap oo wrap)"
  shows "fix·body = wrap·(fix·body')"
proof -
  from body_body'
  have "unwrap oo body oo wrap oo unwrap = body' oo unwrap oo wrap oo unwrap"
    by (simp add: assoc_oo)
  with wrap_unwrap have "unwrap oo body = body' oo unwrap" by simp
  with unwrap_strict have "unwrap·(fix·body) = fix·body'" by (rule lfp_fusion)
  hence "(wrap oo unwrap)·(fix·body) = wrap·(fix·body')" by simp
  with wrap_unwrap show ?thesis by simp
qed
```

A more general result makes fusion optional for each recursive call:

```
lemma worker_wrapper_fusion_new_general:
  fixes wrap :: "'b::pcpo → 'a::pcpo"
  fixes unwrap :: "'a → 'b"
  assumes wrap_unwrap: "wrap oo unwrap = (ID :: 'a → 'a)"
  assumes unwrap_strict: "unwrap·⊥ = ⊥"
  assumes body_body': "⋀r. (unwrap oo wrap)·r = r
                         ⟹ (unwrap oo body oo wrap)·r = body'·r"
  shows "fix·body = wrap·(fix·body')"
```

For a recursive definition $comp = body$ of type A and a pair of functions $wrap :: B \to A$ and $unwrap :: A \to B$ where $wrap \circ unwrap = id_A$ and $unwrap \perp = \perp$, define:

$$\begin{aligned} comp &= wrap\ work \\ work &= unwrap\ (body[wrap\ work / comp]) \end{aligned} \qquad \text{(the worker/wrapper transformation)}$$

In the scope of *work*, the following rewrite is admissable:

$$unwrap\ (wrap\ work) \Longrightarrow work \qquad \text{(worker/wrapper fusion)}$$

Figure B.2: The syntactic worker/wrapper transformation and fusion rule.

```
proof -
  let ?P = "λ(x, y). x = y ∧ unwrap·(wrap·x) = x"
  have "?P (fix·(unwrap oo body oo wrap), (fix·body'))"
  proof(induct rule: parallel_fix_ind)
    case 2 with retraction_strict unwrap_strict wrap_unwrap show "?P (⊥, ⊥)"
      by (bestsimp simp add: cfun_eq_iff)
    case (3 x y) hence xy: "x = y" and unwrap_wrap: "unwrap·(wrap·x) = x" by auto
    from body_body' xy unwrap_wrap
    have "(unwrap oo body oo wrap)·x = body'·y" by simp
    moreover from wrap_unwrap
    have "unwrap·(wrap·((unwrap oo body oo wrap)·x)) = (unwrap oo body oo wrap)·x"
      by (simp add: cfun_eq_iff)
    ultimately show ?case by simp
  qed simp
  thus ?thesis using worker_wrapper_id[OF wrap_unwrap refl] by simp
qed
```

This justifies the syntactically-oriented rules shown in Figure B.2; note the scoping of the fusion rule. Those familiar with the “bananas” work of Meijer, Fokkinga, and Paterson (1991) will not be surprised that adding a strictness assumption justifies an equational fusion rule.

B.4 Backtracking using lazy lists and continuations

We illustrate our worker/wrapper fusion rule by applying it to the first-order part of a higher-order backtracking language by Wand and Vaillancourt (2004); see also Danvy, Grobauer, and Rhiger (2001). We refer the reader to these papers for a broader motivation for these languages.

We use a HOL datatype to define the syntax of our language:

```
datatype expr = const nat | add expr expr | disj expr expr | fail
```

The language consists of constants, an addition function, a disjunctive choice between expressions, and failure. We give it a direct semantics using the monad of lazy lists of natural numbers, with the goal of deriving an an extensionally-equivalent evaluator that uses double-barrelled continuations.

Our theory of lazy lists is entirely standard.

```
default_sort predomain
domain 'a llist = lnil | lcons (lazy "'a") (lazy "'a llist")
```

By relaxing the default sort of type variables to predomain, our polymorphic definitions can be used at concrete types that do not contain ⊥. These include those constructed from HOL types using the discrete ordering, where `x ⊑ y` iff `x = y`.

The following standard list functions underpin the monadic infrastructure:

```
fixrec lappend :: "'a llist → 'a llist → 'a llist" where
  "lappend·lnil·ys = ys"
| "lappend·(lcons·x·xs)·ys = lcons·x·(lappend·xs·ys)"

fixrec lconcat :: "'a llist llist → 'a llist" where
  "lconcat·lnil = lnil"
| "lconcat·(lcons·x·xs) = lappend·x·(lconcat·xs)"

fixrec lmap :: "('a → 'b) → 'a llist → 'b llist" where
  "lmap·f·lnil = lnil"
| "lmap·f·(lcons·x·xs) = lcons·(f·x)·(lmap·f·xs)"
```

We define the lazy list monad S in the traditional fashion:

```
type_synonym S = "nat llist"

definition returnS :: "nat → S" where "returnS = (Λ x. lcons·x·lnil)"
definition bindS :: "S → (nat → S) → S" where "bindS = (Λ x g. lconcat·(lmap·g·x))"
```

The evaluator uses the following extra constants:

```
definition addS :: "S → S → S" where
  "addS ≡ (Λ x y. bindS·x·(Λ xv. bindS·y·(Λ yv. returnS·(xv + yv))))"
definition disjS :: "S → S → S" where "disjS ≡ lappend"
definition failS :: "S" where "failS ≡ lnil"
```

We interpret our language using these combinators in the obvious way. The only complication is that we must explicitly use the fixed point operator as the worker/wrapper technique requires us to talk about the body of the recursive definition.

```
definition evalS_body :: "(expr → nat llist) → (expr → nat llist)" where
  "evalS_body ≡ Λ r e. case e of
     const n ⇒ returnS·n
   | add e1 e2 ⇒ addS·(r·e1)·(r·e2)
   | disj e1 e2 ⇒ disjS·(r·e1)·(r·e2)
   | fail ⇒ failS"
```

```
abbreviation evalS :: "expr → nat llist" where "evalS ≡ fix·evalS_body"
```

We transform this evaluator into one using double-barrelled continuations; one will serve as a "success" context, taking a natural number into "the rest of the computation", and the other failure.

In general we could work with an arbitrary observation type ala Reynolds (1974), but for convenience we use the clearly adequate concrete type `nat llist`.

```
type_synonym Obs = "nat llist"
type_synonym Failure = "Obs"
type_synonym Success = "nat → Failure → Obs"
type_synonym K = "Success → Failure → Obs"
```

To ease our development we adopt what Wand and Vaillancourt (2004, §5) call a "failure computation" instead of a failure continuation, which would have the type `unit → Obs`.

The monad over the continuation type K is as follows:

```
definition returnK :: "nat → K" where "returnK ≡ (Λ x. Λ s f. s·x·f)"
definition bindK :: "K → (nat → K) → K" where
  "bindK ≡ Λ x g. Λ s f. x·(Λ xv f'. g·xv·s·f')·f"
```

Our extra constants are defined as follows:

```
definition addK :: "K → K → K" where
  "addK ≡ (Λ x y. bindK·x·(Λ xv. bindK·y·(Λ yv. returnK·(xv + yv))))"
definition disjK :: "K → K → K" where "disjK ≡ (Λ g h. Λ s f. g·s·(h·s·f))"
definition failK :: "K" where "failK ≡ Λ s f. f"
```

The continuation semantics is again straightforward:

```
definition evalK_body :: "(expr → K) → (expr → K)" where
  "evalK_body ≡ Λ r e. case e of
     const n ⇒ returnK·n
   | add e1 e2 ⇒ addK·(r·e1)·(r·e2)
   | disj e1 e2 ⇒ disjK·(r·e1)·(r·e2)
   | fail ⇒ failK"
```

```
abbreviation evalK :: "expr → K" where "evalK ≡ fix·evalK_body"
```

We now set up a worker/wrapper relation between these two semantics.

The kernel of `unwrapB` is the following function that converts a lazy list into an equivalent continuation representation.

```
fixrec SK :: "S → K" where
  "SK·lnil = failK"
| "SK·(lcons·x·xs) = (Λ s f. s·x·(SK·xs·s·f))"
```

```
definition unwrapB :: "(expr → nat llist) → (expr → K)" where
  "unwrapB ≡ Λ r e. SK·(r·e)"
```

Symmetrically wrapB converts an evaluator using continuations into one generating lazy lists by passing it the right continuations.

```
definition KS :: "K → S" where "KS ≡ Λ k. k·lcons·lnil"
definition wrapB :: "(expr → K) → (expr → nat llist)" where
  "wrapB ≡ Λ r e. KS·(r·e)"
```

The worker/wrapper condition follows directly from these definitions.

```
lemma KS_SK_id: "KS·(SK·xs) = xs"
  by (induct xs) (simp_all add: KS_def failK_def)

lemma wrapB_unwrapB_id: "wrapB oo unwrapB = ID"
  unfolding wrapB_def unwrapB_def
  by (simp add: KS_SK_id cfun_eq_iff)
```

The worker/wrapper transformation is only non-trivial if wrapB and unwrapB do not witness an isomorphism. In this case we can show that we do not even have a Galois connection.

```
lemma cfun_not_below: "f·x ⋢ g·x ⟹ f ⋢ g" by (auto simp: cfun_below_iff)

lemma unwrapB_wrapB_not_below_id: "unwrapB oo wrapB ⋢ ID"
proof -
  let ?witness = "Λ e. (Λ s f. lnil :: K)"
  have "(unwrapB oo wrapB)·?witness·fail·⊥·(lcons·0·lnil)
       ⋢ ?witness·fail·⊥·(lcons·0·lnil)"
    by (simp add: failK_def wrapB_def unwrapB_def KS_def)
  hence "(unwrapB oo wrapB)·?witness ⋢ ?witness"
    by (fastforce intro!: cfun_not_below)
  thus ?thesis by (simp add: cfun_not_below)
qed
```

We now apply `worker_wrapper_id`:

```
definition eval_work :: "expr → K" where
  "eval_work ≡ fix·(unwrapB oo evalS_body oo wrapB)"

definition eval_ww :: "expr → nat llist" where
  "eval_ww ≡ wrapB·eval_work"

lemma "evalS = eval_ww"
  unfolding eval_ww_def eval_work_def
  using worker_wrapper_id[OF wrapB_unwrapB_id] by simp
```

We now show how the monadic operations correspond by showing that SK witnesses a *monad morphism* (Wadler 1992, §6). As required by Danvy et al. (2001, Definition 2.1), the mapping needs to hold for our specific operations in addition to the common monadic scaffolding.

```
lemma SK_returnS_returnK: "SK·(returnS·x) = returnK·x"
  by (simp add: returnS_def returnK_def failK_def)

lemma SK_lappend_distrib: "SK·(lappend·xs·ys)·s·f = SK·xs·s·(SK·ys·s·f)"
  by (induct xs) (simp_all add: failK_def)

lemma SK_bindS_bindK: "SK·(bindS·x·g) = bindK·(SK·x)·(SK oo g)"
  by (induct x)
     (simp_all add: cfun_eq_iff bindS_def bindK_def failK_def SK_lappend_distrib)

lemma SK_addS_distrib: "SK·(addS·x·y) = addK·(SK·x)·(SK·y)"
  by (clarsimp simp: cfcomp1 addS_def addK_def failK_def
                     SK_bindS_bindK SK_returnS_returnK)

lemma SK_disjS_disjK: "SK·(disjS·xs·ys) = disjK·(SK·xs)·(SK·ys)"
  by (simp add: cfun_eq_iff disjS_def disjK_def SK_lappend_distrib)

lemma SK_failS_failK: "SK·failS = failK"
  unfolding failS_def by simp
```

These lemmas establish the precondition for our all-in-one worker/wrapper and fusion rule:

```
lemma evalS_body_evalK_body:
  "unwrapB oo evalS_body oo wrapB = evalK_body oo unwrapB oo wrapB"
proof(intro cfun_eqI)
  fix r e s f show "(unwrapB oo evalS_body oo wrapB)·r·e·s·f
                  = (evalK_body oo unwrapB oo wrapB)·r·e·s·f"
    by (cases e) (simp_all add: evalS_body_def evalK_body_def unwrapB_def
                                SK_returnS_returnK SK_addS_distrib
                                SK_disjS_disjK SK_failS_failK)
qed

theorem evalS_evalK: "evalS = wrapB·evalK"
  using worker_wrapper_fusion_new[OF wrapB_unwrapB_id unwrapB_strict]
        evalS_body_evalK_body by simp
```

This proof can be considered an instance of the approach of Hutton, Jaskelioff, and Gill (2010), which uses the worker/wrapper machinery to relate two algebras.

This result could be obtained by a structural induction over the syntax of the language. A strength of our proof however is that it can be extended, e.g. to the full language of Danvy et al. (2001) simply by proving extra equations. In contrast the higher-order language of Wand and Vaillancourt (2004) is beyond the reach of this approach.

Bibliography

M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

A. Abel. MiniAgda: Integrating sized and dependent types. In A. Bove, E. Komendantskaya, and M. Niqui, editors, *Proceedings of the Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR '10)*, volume 43 of *EPTCS*, pages 14–28, 2010.

A. V. Aho, J. D. Ullman, and M. Yannakakis. Modeling communications protocols by automata. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS '79)*, pages 267–273, 1979. IEEE Computer Society, Washington, DC.

O. I. Al-Bataineh and R. van der Meyden. Epistemic model checking for knowledge-based program implementation: An application to anonymous broadcast. In S. Jajodia and J. Zhou, editors, *SecureComm*, volume 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 429–447, 2010. Springer, Berlin.

O. I. Al-Bataineh and R. van der Meyden. Abstraction for epistemic model checking of dining cryptographers-based protocols. In K. R. Apt, editor, *Proceedings of the 13th Conference on Theoretical Aspects of Rationality and Knowledge (TARK '11)*, pages 247–256, 2011. ACM, New York, NY.

J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In T. Touili, B. Cook, and P. Jackson, editors, *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV '10)*, volume 6174 of *LNCS*, pages 258–272, 2010. Springer, Berlin.

R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, 160(1-2):167–188, 2000.

C. André and M.-A. Peraldi-Frati. Behavioral specification of a circuit using SyncCharts: A case study. In *26th EUROMICRO 2000 Conference, Informatics: Inventing the Future (EUROMICRO '00)*, page 1091, 2000. IEEE Computer Society, Washington, DC.

J. K. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

Arvind and R. S. Nikhil. Hands-on introduction to Bluespec System Verilog (BSV) (abstract). In *Proceedings of 6th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '08)*, pages 205–206, 2008. IEEE Computer Society, Washington, DC.

P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems*, 26(1):125–185, 2004.

R.J. Aumann. Agreeing to disagree. *The Annals of Statistics*, 4(6):1236–1239, 1976.

K. Avnit, V. D'Silva, A. Sowmya, S. Ramesh, and S. Parameswaran. Provably correct on-chip communication: A formal approach to automatic protocol converter synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 14(2), 2009.

B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985.

E. Axelsson, K. Claessen, and M. Sheeran. Wired: Wire-aware circuit design. In D. Borrione and W. J. Paul, editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference (CHARME '05)*, volume 3725 of *LNCS*, pages 5–19, 2005. Springer, Berlin.

E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of Feldspar - an embedded language for digital signal processing. In J. Hage and M. T. Morazán, editors, *Selected papers from 22nd International Symposium on Implementation and Application of Functional Languages (IFL '10)*, volume 6647 of *LNCS*, pages 121–136, 2010. Springer, Berlin.

Emil Axelsson. Description and analysis of multipliers using Lava. Master's thesis, Chalmers University of Technology, 2003.

J. W. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. M. Borwein and W. M. Farmer, editors, *Proceedings of the 5th International Conference on Mathematical Knowledge Management (MKM '06)*, volume 4108 of *LNCS*, pages 31–43, 2006. Springer, Berlin.

A. Baltag and L. S. Moss. Logics for epistemic programs. *Synthese*, 139(2):165 – 224, 2004.

G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

K. Baukus and R. van der Meyden. A knowledge-based analysis of cache coherence. In J. Davies, W. Schulte, and M. Barnett, editors, *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM '04)*, volume 3308 of *LNCS*, pages 99–114, 2004. Springer, Berlin.

H. Bekić. Definable operations in general algebras, and the theory of automata and flowcharts. In *Programming Languages and Their Definition - Hans Bekić (1936-1982)*, pages 30–55, 1984. Springer, Berlin.

S. Bensalem, D. Peled, and J. Sifakis. Knowledge based scheduling of distributed systems. In Z. Manna and D. Peled, editors, *Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*, pages 26–41. Springer, 2010.

A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

S. Berghofer and M. Reiter. Formalizing the logic-automaton connection. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 147–163, 2009. Springer, Berlin.

G. Berry. Real time programming: Special purpose or general purpose languages? In G. Ritter, editor, *Proceedings of the IFIP 11th World Computer Congress (Information Processing '89)*, pages 11–17, 1989. North-Holland/IFIP, Amsterdam.

G. Berry. The constructive semantics of pure Esterel. Draft book (version 3), 1999a.

G. Berry. The Esterel v5 language primer. Draft book, 1999b.

Y. Bertot and E. Komendantskaya. Using structural recursion for corecursion. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *Revised Selected Papers from the International Conference on Types for Proofs and Programs (TYPES '08)*, volume 5497 of *LNCS*, pages 220–236, 2008. Springer, Berlin.

Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors. *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *LNCS*, 1999. Springer, Berlin.

M. Bickford, R. L. Constable, J. Y. Halpern, and S. Petride. Knowledge-based synthesis of distributed systems using event structures. *Logical Methods in Computer Science*, 7(2), 2009.

A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *LNCS*, pages 193–207, 1999. Springer, Berlin.

R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer, Berlin, 1987. NATO ASI Series F Volume 36.

R. S. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, 1998. ACM, New York, NY.

B. Bose and S. D. Johnson. DDD-FM9001: Derivation of a verified microprocessor. In G. J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference (CHARME '93)*, volume 683 of *LNCS*, pages 191–202, 1993. Springer, Berlin.

A. Bouajjani, J.-C. Fernandez, N. Halbwachs, and P. Raymond. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–269, 1992.

R. J. Boulton, A. D. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience (TPCD '92)*, volume A-10 of *IFIP Transactions*, pages 129–156, 1992. North-Holland, Amsterdam.

F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjørner, M. Broy, and I. Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *LNCS*, pages 128–141. Springer, 1993.

F. Bourdoncle and S. Merz. Type-checking higher-order polymorphic multi-methods. In P. Lee, F. Henglein, and N. D. Jones, editors, *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 302–315, 1997. ACM, New York, NY. ISBN 0-89791-853-3.

T. Bourke. *Modelling and Programming Embedded Controllers with Timed Automata and Synchronous Languages.* PhD thesis, CSE, UNSW, 2009.

R. I. Brafman, J-C. Latombe, Y. Moses, and Y. Shoham. Applications of a logic of knowledge to motion planning under uncertainty. *Journal of the ACM*, 44(5), 1997.

T. Braibant. Coquet: A Coq library for verifying hardware. In J.-P. Jouannaud and Z. Shao, editors, *Proceedings of the 1st International Conference on Certified Programs and Proofs (CPP '11)*, volume 7086 of *LNCS*, pages 330–345, 2011. Springer, Berlin.

S. D. Brookes. Historical introduction to "Concrete Domains" by G. Kahn and G. D. Plotkin. *Theoretical Computer Science*, 121(1&2):179–186, 1993.

F. P. Brooks Jr. *The mythical man-month – essays on software engineering.* Addison-Wesley, Reading, Massachusetts, second edition, 1995.

S. Browning and P. Weaver. Designing tunable, verifiable cryptographic hardware using Cryptol. In Hardin (2010).

M. Broy and K. Stølen. *Specification and development of interactive systems: FOCUS on streams, interfaces, and refinement.* Monographs in Computer Science. Springer, Berlin, 2001.

R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2): 205–213, 1991.

Janusz A. Brzozowski and Carl-Johan Seger. *Asynchronous Circuits.* Springer, Berlin, 1995.

J. R. Burch, D. L. Dill, E. Wolf, and G. De Micheli. Modeling hierarchical combinational circuits. In M. R. Lightner and J. A. G. Jess, editors, *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '93)*, pages 612–617, 1993. IEEE Computer Society, Washington, DC.

R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

G. C. Buttazzo, editor. *Proceedings of the 4th ACM International Conference On Embedded Software (EMSOFT '04)*, 2004. ACM, New York, NY.

A. Camilleri, M. Gordon, and T. Melham. Hardware verification using Higher-Order Logic. Technical Report 91, Computer Laboratory, University of Cambridge, 1986.

L. Cardelli. *An Algebaic Approach to Hardware Description and Verification.* PhD thesis, University of Edinburgh, 1982.

L. Cardelli and G. D. Plotkin. An algebraic approach to VLSI design. In J. P. Gray, editor, *VLSI*, 1981. Academic Press, New York, NY.

J. M. P. Cardoso, P. C. Diniz, and M. Weinhardt. Compiling for reconfigurable computing: A survey. *ACM Computing Surveys*, 42(4), 2010.

J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94(1):125–140, 1992.

P. Caspi and M. Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. *Electronic Notes on Theoretical Computer Science*, 11:1–21, 1998.

M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, 2005. ACM, New York, NY.

M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors. *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*, 2011a. ACM, New York, NY.

M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In M. Carro and J. H. Reppy, editors, *Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP '11)*, pages 3–14, 2011b. ACM, New York, NY.

A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

D. Chaum. The dining cryptographers problem: unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

B. Chellas. *Modal Logic: an introduction*. Cambridge University Press, 1980.

Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFAs for compositional verification. In S. Kowalewski and A. Philippou, editors, *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '09)*, volume 5505 of *LNCS*, pages 31–45, 2009. Springer, Berlin.

A. Cimatti, C. Pecheur, and R. Cavada. Formal verification of diagnosability via symbolic model checking. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI '03)*, pages 363–369. Morgan Kaufmann, 2003.

A. Cimatti, C. Pecheur, and A. Lomuscio. Applications of model checking for multi-agent systems: Verification of diagnosability and recoverability. In *Proceedings of the International Workshop on Concurrency, Specification, and Programming*, 2005.

K. Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, 2001.

K. Claessen. Safety property verification of cyclic synchronous circuits. In *Proceedings of Workshop on Synchronous Languages Applications and Programs (SLAP)*, volume 88 of *Electronic Notes on Theoretical Computer Science*, 2003. Elsevier, Amsterdam.

K. Claessen, M. Sheeran, and S. Singh. Using Lava to design and verify recursive and periodic sorters. *International Journal on Software Tools for Technology Transfer*, 4(3):349–358, 2003.

E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6 (2):217–232, 1995.

E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

M. Cohen, M. Dam, A. Lomuscio, and H. Qu. A symmetry reduction technique for model checking temporal-epistemic logic. In C. Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, pages 721–726, 2009.

J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous dataflow language. In Buttazzo (2004).

T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *International Workshop on Types for Proofs and Programs (TYPES '93)*, volume 806 of *LNCS*, pages 62–78, 1993. Springer, Berlin.

P.-L. Curien. *Categorical combinators, sequential algorithms, and functional programming*. Birkhauser Boston, Cambridge, MA, USA, second edition, 1994.

N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In J. G. Morrisett and S. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*, pages 206–217, 2006. ACM, New York, NY.

O. Danvy, B. Grobauer, and M. Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1):53–74, 2001.

N. A. Day, M. Aagaard, and B. Cook. Combining stream-based and state-based verification techniques. In W. A. Hunt Jr. and S. D. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD '00)*, volume 1954 of *LNCS*, pages 126–142, 2000. Springer, Berlin.

J. W. de Bakker, A. de Bruin, and J. Zucker. *Mathematical theory of program correctness*. Prentice-Hall, 1980.

H. W. de Haan, W. H. Hesselink, and G. R. R. de Lavalette. Knowledge-based asynchronous programming. *Fundamenta Informaticae*, 63(2-3):259–281, 2004.

W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, U.K., 1998.

I. S. Diatchki, M. P. Jones, and R. Leslie. High-level views on low-level representations. In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*, pages 168–179, 2005. ACM, New York, NY.

C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

K. Engelhardt, R. van der Meyden, and Y. Moses. A refinement theory that supports reasoning about knowledge and time. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '01)*, volume 2250 of *LNCS*, pages 125–141, 2001. Springer, Berlin.

K. Engelhardt, P. Gammie, and R. van der Meyden. Model checking knowledge and linear time: PSPACE cases. In S. N. Artëmov and A. Nerode, editors, *Proceedings of the International Symposium on the Logical Foundations of Computer Science (LFCS '07)*, volume 4514 of *LNCS*, pages 195–211, 2007. Springer, Berlin.

L. Erkök. *Value recursion in monadic computations.* PhD thesis, OGI School of Science and Engineering, OHSU, Portland, Oregon, 2002.

M. Ersan and V. Akman. Situated modeling of epistemic puzzles. *Logic Journal of the IGPL*, 3(1): 51–76, 1995.

R. Fagin, J. Y. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge.* The MIT Press, 1995.

R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.

K. Fisler and M. Y. Vardi. Bisimulation and model checking. In Laurence Pierre and Thomas Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*, volume 1703 of *LNCS*, pages 338–341, 1999. Springer, Berlin.

M. M. Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, 1991.

A. C. J. Fox, M. J. C. Gordon, and M. O. Myreen. Specification and verification of ARM hardware and software. In Hardin (2010).

S. Frankau and A. Mycroft. Stream processing hardware from functional language specifications. In *36th Hawaii International Conference on System Sciences (HICSS-36 2003)*, page 278, 2003. IEEE Computer Society, Washington, DC.

D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In *Third International Colloquium on Automata, Languages and Programming (ICALP '76)*, pages 257–284, 1976. Edinburgh University Press, Edinburgh, UK.

P. Gammie. The worker/wrapper transformation. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/WorkerWrapper.shtml`, 2009. Formal proof development.

P. Gammie. Knowledge-based programs. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/KBPs.shtml`, 2011a. Formal proof development.

P. Gammie. Verified synthesis of knowledge-based programs in finite synchronous environments. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Proceedings of the Second International Conference on Interactive Theorem Proving (ITP '11)*, volume 6898 of *LNCS*, pages 87–102. Springer, 2011b.

P. Gammie. Short note: Strict unwraps make worker/wrapper fusion totally correct. *Journal of Functional Programming*, 21(2):209–213, 2011c.

P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *LNCS*, pages 479–483, 2004. Springer, Berlin.

J. Geldenhuys and A. Valmari. Techniques for smaller intermediary BDDs. In K. G. Larsen and M. Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR '01)*, volume 2154 of *LNCS*, pages 233–247, 2001. Springer, Berlin.

D. R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*, pages 363–375, 2007. ACM, New York, NY.

D. R. Ghica, A. Smith, and S. Singh. Geometry of synthesis IV: compiling affine recursion into static hardware. In Chakravarty, Hu, and Danvy (2011a), pages 221–233.

A. Gill, editor. *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell (Haskell '08)*, 2008. ACM, New York, NY.

A. Gill. Type-safe observable sharing in Haskell. In S. Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*, 2009. ACM, New York, NY.

A. Gill and A. Farmer. Deriving an efficient FPGA implementation of a low density parity check forward error corrector. In Chakravarty et al. (2011a), pages 209–220.

A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, 2009.

J. Gillenwater, G. Malecha, C. Salama, A. Yun Zhu, W. Taha, J. Grundy, and J. O'Leary. Synthesizable high level hardware descriptions. *New Generation Computing*, 28(4):339–369, 2010.

A. Girault and E. Rutten. Automating the addition of fault tolerance with discrete event controller synthesis. *Formal Methods in System Design*, 35:190–225, 2009.

M. J. C. Gordon. The semantic challenge of Verilog HDL. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science (LICS '95)*, pages 136–145, 1995. IEEE Computer Society, Washington, DC.

S. Greibach. *Theory of program structures: schemes, semantics, verification*, volume 36 of *LNCS*. Springer, Berlin, 1975.

D. Gries. Describing an algorithm by Hopcroft. *Acta Informaticae*, 2:97–109, 1973.

G. Grov and G. Michaelson. Hume box calculus: robust system development through software transformation. *Higher-Order and Symbolic Computation*, 23(2):191–226, 2010.

J. Grundy, T. F. Melham, and J. W. O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, Cambridge, MA, USA, 1992.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST '93)*, Workshops in Computing, pages 83–96, 1993. Springer, Berlin.

J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. *Journal of the ACM*, 37(3), 1990.

J. Y. Halpern and K. R. O'Neill. Anonymity and information hiding in multiagent systems. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW '03)*, 2003. IEEE Computer Society, Washington, DC.

J. Y. Halpern and L. D. Zuck. A little knowledge goes a long way: Knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, 1992.

G. Hamon. Synchronous dataflow pattern matching. *Electronic Notes on Theoretical Computer Science*, 153(4):37–54, 2006.

J. Handy. *The cache memory book - the authoritative reference on cache design.* Academic Press, New York, NY, second edition, 1998.

F. K. Hanna. Reasoning about analog-level implementations of digital systems. *Formal Methods in System Design*, 16(2):127–158, 2000.

D. S. Hardin, editor. *Design and Verification of Microprocessor Systems for High-Assurance Applications*, 2010. Springer, Berlin.

D. Harel. On folk theorems. *Communications of the ACM*, 23(7):379–389, 1980.

D. Harel. Statecharts in the making: a personal account. *Communications of the ACM*, 52(3): 67–75, 2009.

W. L. Harrison, A. M. Procter, J. Agron, G. Kimmell, and G. Allwein. Model-driven engineering from modular monadic semantics: Implementation techniques targeting hardware and software. In W. Taha, editor, *Domain-Specific Languages, IFIP TC 2 Working Conference (DSL '09)*, volume 5658 of *LNCS*, 2009. Springer, Berlin.

P. Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming (LFP '82)*, pages 179–187, 1982. ACM, New York, NY.

J. Hintikka. *Knowledge and Belief: An Introduction to the Logic of Two Notions.* Cornell University Press, 1962.

R. Hinze and Daniel W. H. James. Proving the unique fixed-point principle correct: an adventure with category theory. In Chakravarty et al. (2011a), pages 359–371.

G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5): 279–295, 1997.

A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In A. E. Dunlop, editor, *Proceedings of the 30th Design Automation Conference (DAC '93)*, pages 266–271. ACM/IEEE, 1993.

P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, 1996.

P. Hudak and S. Weirich, editors. *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*, 2010. ACM, New York, NY.

B. Huffman. *HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs*. PhD thesis, Portland State University, 2012.

J. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Programming Research Group, Oxford University, July 1983.

J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

J. Hughes, editor. *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '91)*, volume 523 of *LNCS*, 1991. Springer, Berlin.

J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, 2000.

J. Hughes. Programming with Arrows. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *LNCS*, pages 73–129. Springer, 2004.

J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 410–423, 1996. ACM, New York, NY.

W. A. Hunt Jr., S. Swords, J. Davis, and A. Slobodova. Use of formal verification at Centaur Technology. In Hardin (2010).

G. Hutton, M. Jaskelioff, and A. Gill. Factorising folds for faster functions. *Journal of Functional Programming*, 20(3-4):353–373, 2010.

A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, 2005.

S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. ACM Distinguished Dissertation Series. MIT Press, Cambridge, MA, 1983.

S. D. Johnson. View from the fringe of the fringe. In T. Margaria and T. F. Melham, editors, *Proceedings of Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference (CHARME '01)*, volume 2144 of *LNCS*, pages 1–12, 2001. Springer, Berlin.

S. D. Johnson and B. Bose. DDD: A system for mechanized digital design derivation. Technical Report 323, Indiana University, 1997.

G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Proceedings of the 2nd International Conference on Mathematics of Program Construction (MPC '93)*, pages 208–232, 1993. Springer, Berlin.

M. P. Jones. Type classes with functional dependencies. In Gert Smolka, editor, *Proceedings of the 9th European Symposium on Programming (ESOP '00)*, volume 1782 of *LNCS*, pages 230–244, 2000. Springer, Berlin.

M. P. Jones and I. S. Diatchki. Language and program design for functional dependencies. In Gill (2008), pages 87–98.

M. Kacprzak, A. Lomuscio, A. Niewiadomski, W. Penczek, F. Raimondi, and M. Szreter. Comparing BDD and SAT based techniques for model checking Chaum's Dining Cryptographers protocol. *Fundamenta Informaticae*, 72(1-3):215–234, 2006.

M. Kacprzak, W. Nabialek, A. Niewiadomski, W. Penczek, A. Pólrola, M. Szreter, B. Wozna, and A. Zbrzezny. VerICS 2007 - a model checker for knowledge and real-time. *Fundamenta Informaticae*, 85(1-4):313–328, 2008.

S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming (ESOP '88)*, volume 300 of *LNCS*, pages 131–144, 1988. Springer, Berlin.

G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Proceedings of IFIP Congress 1974 (Information Processing '74)*, 1974. North-Holland, Amsterdam.

F. Kammüller, M. Wenzel, and L. C. Paulson. Locales - a sectioning concept for isabelle. In Bertot, Dowek, Hirschowitz, Paulin, and Théry (1999), pages 149–166.

G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In Hudak and Weirich (2010), pages 261–272.

O. Kiselyov. Solving the Mr. S and Mr. P puzzle by John McCarthy, 2006. URL `http://okmij.org/ftp/Haskell/Mr-S-P.lhs`.

O. Kiselyov. Implementing explicit and finding implicit sharing in embedded DSLs. In O. Danvy and C.-C. Shan, editors, *Proceedings of the IFIP Working Conference on Domain-Specific Languages (DSL '11)*, volume 66 of *EPTCS*, pages 210–225, 2011.

O. Kiselyov, K. N. Swadi, and W. Taha. A methodology for generating verified combinatorial circuits. In Buttazzo (2004).

C. D. Kloos. *Semantics of Digital Circuits*, volume 285 of *LNCS*. Springer, Berlin, 1987.

N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-order functional reactive programming in bounded space. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*, pages 45–58, 2012. ACM, New York, NY.

P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In M. Kaufmann and L. C. Paulson, editors, *Proceedings of the 1st International Conference on Interactive Theorem Proving (ITP '10)*, volume 6172 of *LNCS*, pages 339–354, 2010. Springer, Berlin.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4): 293–341, 1995.

J. Launchbury, J. R. Lewis, and B. Cook. On embedding a microarchitectural design language within Haskell. In *Proceedings of the 4th International Conference on Functional Programming (ICFP '99)*, pages 60–69, 1999. ACM, New York, NY.

D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC '92)*, pages 264–274, 1992. ACM, New York, NY.

W. Lenzen. Recent work in epistemic logic. *Acta Philosophica Fennica*, 30(1), 1978.

D. K. Lewis. *Convention: a philosophical study*. Harvard University Press, 1969.

J. R. Lewis, J. Launchbury, E. Meijer, and M. Shields. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL '00)*, pages 108–118, 2000. ACM, New York, NY.

P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411 (19):1974–1994, 2010.

O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL '85)*, pages 97–107, 1985.

A. Lomuscio and F. Raimondi. The complexity of model checking concurrent programs against CTLK specifications. In H. Nakashima, M. P. Wellman, G. Weiss, and P. Stone, editors, *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*, pages 548–550, 2006a. ACM, New York, NY.

A. Lomuscio and F. Raimondi. MCMAS: A model checker for multi-agent systems. In H. Hermanns and J. Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, volume 3920 of *LNCS*, pages 450–454, 2006b. Springer, Berlin.

A. Lomuscio, R. van der Meyden, and M. Ryan. Knowledge in multiagent systems: initial configurations and broadcast. *ACM Transactions on Computational Logic*, 1(2):247–284, 2000.

A. Lomuscio, F. Raimondi, and B. Wozna. Verification of the TESLA protocol in MCMAS-X. *Fundamenta Informaticae*, 79(3-4):473–486, 2007.

A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In A. Bouajjani and O. Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*, volume 5643 of *LNCS*, pages 682–688, 2009. Springer, Berlin.

X. Luo, K. Su, A. Sattar, and Y. Chen. Solving sum and product riddle via BDD-based model checking. In *Web Intelligence/IAT Workshops*, pages 630–633. IEEE, 2008.

X. Luo, K. Su, M. Gu, L. Wu, and J. Yang. Symbolic model checking the knowledge in Herbivore protocol. In R. van der Meyden and J.-G. Smaus, editors, *Revised Selected and Invited Papers from the 6th International Workshop on Model Checking and Artificial Intelligence (MoChArt '10)*, volume 6572 of *LNCS*, pages 112–129, 2010. Springer, Berlin.

N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

S. Malik. Analysis of cyclic combinational circuits. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '93)*, pages 618–625, 1993. IEEE Computer Society, Washington, DC.

Z. Manna. *Introduction to Mathematical Theory of Computation*. McGraw-Hill, Inc., New York, NY, USA, 1974.

Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, Berlin, 1992.

F. Maraninchi and N. Halbwachs. Compositional semantics of non-deterministic synchronous languages. In H. R. Nielson, editor, *6th European Symposium on Programming (ESOP '96)*, volume 1058 of *LNCS*, pages 235–249, 1996. Springer, Berlin.

F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1/3):61–92, 2001.

J. Matthews. Recursive function definition over coinductive types. In Bertot et al. (1999), pages 73–90.

J. Matthews. *Algebraic Specification and Verification of Processor Microarchitectures*. PhD thesis, Oregon Graduate Institute of Science and Technology, 2000.

J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages (ICCL '98)*, page 90, 1998. IEEE Computer Society, Washington, DC.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

J. McCarthy. Formalization of two Puzzles Involving Knowledge, 1987. URL `http://www-formal.stanford.edu/jmc/puzzles.html`.

K. McEvoy and J. V. Tucker. Theoretical foundations of hardware design. In McEvoy and Tucker (1990b), chapter 1.

K. McEvoy and J.V. Tucker, editors. *Theoretical Foundations of VLSI Design*, number 10 in Cambridge Tracts in Theoretical Computer Science, 1990b. Cambridge University Press, Cambridge, UK.

C. Mead and L. Conway. *Introduction to VLSI systems*. Addison-Wesley, Reading, MA, 1980.

A. Megacz. Multi-level languages are Generalized Arrows. *CoRR*, abs/1007.2885, 2010.

A. Megacz. Hardware design with Generalized Arrows. Technical Report ITTC-FY2012-TR-29952012-01 (IFL 2011 Draft Proceedings), University of Kansas, 2011.

E. Meijer. *Calculating Compilers*. PhD thesis, Katholieke Univeristeit Nijmegen, 1992.

E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In Hughes (1991), pages 124–144.

M. Mendler, T. R. Shiple, and G. Berry. Constructive Boolean circuits and the exactness of timed ternary simulation. *Formal Methods in System Design*, 40(3):283–329, 2012.

M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

G. J. Milne. CIRCAL and the representation of communication, concurrency, and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, 1985.

G. J. Milne. Modelling dynamically changing hardware structure. *Electronic Notes on Theoretical Computer Science*, 162:249–254, 2006.

R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

B. Möller and J. V. Tucker, editors. *Prospects for Hardware Foundations, ESPRIT Working Group 8533, NADA - New Hardware Design Methods, Survey Chapters*, volume 1546 of *LNCS*, 1998. Springer, Berlin.

A. K. Moran. *Call-by-name, Call-by-need and McCarthy's Amb*. PhD thesis, Department of Computing Science, Chalmers University of Technology, 1998.

Y. Moses, D. Dolev, and J. Y. Halpern. Cheating husbands and other stories: A case study of knowledge, action, and communication. *Distributed Computing*, 1(3):167–176, 1986.

O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

A. Mycroft and R. Sharp. Higher-level techniques for hardware description and synthesis. *International Journal on Software Tools for Technology Transfer*, 4(3):271–297, 2003.

K. S. Namjoshi and R. P. Kurshan. Efficient analysis of cyclic definitions. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99)*, volume 1633 of *LNCS*, pages 394–405, 1999. Springer, Berlin.

M. Naylor and C. Runciman. Expressible sharing for functional circuit description. *Higher-Order and Symbolic Computation*, 22(1):67–80, 2009.

M. Naylor and C. Runciman. The Reduceron reconfigured and re-evaluated. *Journal of Functional Programming*, 22(4-5):574–613, 2012.

O. Neiroukh, S. A. Edwards, and X. Song. Transforming cyclic circuits into acyclic equivalents. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 27(10):1775–1787, 2008.

R. S. Nikhil. Abstraction in hardware system design. *Communications of the ACM*, 54(10):36–44, 2011.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, Berlin, 2002.

M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Conference Record of the SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pages 135–146, 1995.

J. O'Donnell. Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, 1987. North-Holland, Amsterdam.

J. O'Donnell. Generating netlists from executable circuit specifications. In J. Launchbury and P. M. Sansom, editors, *Proceedings of the 1992 Glasgow Workshop on Functional Programming (Functional Programming '92)*, Workshops in Computing, pages 178–194, 1992. Springer, Berlin.

J. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In P. H. Hartel and M. J. Plasmeijer, editors, *Proceedings of 1st International Symposium on Functional Programming Languages in Education (FPLE'95)*, volume 1022 of *LNCS*, pages 195–214, 1995. Springer, Berlin.

J. O'Donnell. Embedding a Hardware Description Language in Template Haskell. In C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 143–164, 2003. Springer, Berlin.

J. O'Donnell and G. Rünger. Derivation of a logarithmic time carry lookahead addition circuit. *Journal of Functional Programming*, 14(6):697–713, 2004.

D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.

S. Park and H. Im. A calculus for hardware description. *Journal of Functional Programming*, 21 (1):21–58, 2011.

R. Paterson. A new notation for Arrows. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*, pages 229–240, 2001. ACM, New York, NY.

R. Paterson. Arrows and Computation. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 201–222. Palgrave Macmillan, New York, NY, 2003.

C. Paulin-Mohring. Circuits as streams in Coq: Verification of a sequential multiplier. In S. Berardi and M. Coppo, editors, *International Workshop on Types for Proofs and Programs (TYPES '95)*, volume 1158 of *LNCS*, pages 216–230, 1995. Springer, Berlin.

S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, Cambridge, U.K., 2003.

S. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In Hughes (1991), pages 636–666.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.

C. P. Pfleeger. State reduction in incompletely specified finite-state machines. *IEEE Transactions on Computers*, C-22(12):1099 – 1102, 1973.

N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '07)*, volume 3855 of *LNCS*, pages 364–380, 2006. Springer, Berlin.

A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

Plato. *Theaetetus*. Penguin classics. Penguin Books, 1987. Translated by R. Waterfield.

J. Plaza. Logics of public communications. *Synthese*, 158(2):165–179, 2007. (originally published in IMIS 1989).

G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5: 223–255, 1977.

G. D. Plotkin. Pisa notes (on domain theory). Unpublished, 1983. URL http://homepages.inf.ed.ac.uk/gdp/publications/Domains.ps.

A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*, pages 179–190, 1989. ACM, New York, NY.

F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, 1996.

D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, Berlin, 2007.

J. C. Reynolds. On the relation between direct and continuation semantics. In J. Loeckx, editor, *Proceedings of the 2nd Colloquium on Automata, Languages and Programming (ICALP '74)*, volume 14 of *LNCS*, pages 141–156, 1974. Springer, Berlin.

J.-K. Rho, G. D. Hachtel, F. Somenzi, and R. M. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(2):167–177, 1994.

S.L. Ricker and K. Rudie. Knowledge is a terrible thing to waste: Using inference in discrete-event control problems. *IEEE Transactions on Automatic Control*, 52(3):428 –441, 2007.

M. D. Riedel and J. Bruck. The synthesis of cyclic combinational circuits. In *Proceedings of the 40th Design Automation Conference (DAC '03)*, pages 163–168, 2003. ACM, New York, NY.

F. Rocheteau and N. Halbwachs. POLLUX: A Lustre based hardware design environment. In P. Quinton and Y. Robert, editors, *Algorithms and Parallel VLSI Architectures*, pages 335–346, 1991. Elsevier, Amsterdam.

S. J. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In J. Y. Halpern, editor, *Proceedings of the 1st Conference on Theoretical Aspects of Reasoning about Knowledge (TARK '86)*, pages 83–98. Morgan Kaufmann, 1986.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In Gill (2008), pages 13–24.

B. Sanders. A predicate transformer approach to knowledge and knowledge-based protocols (extended abstract). In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC '91)*, pages 217–230, 1991. ACM, New York, NY.

D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems*, 31(4), 2009.

W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 341–352. ACM, 2009.

C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. F. Melham, M. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 24(9):1381–1405, 2005.

P. Selinger. A survey of graphical languages for monoidal categories. In Bob Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*. Springer, Berlin, 2011.

R. Sharp and O. Rasmussen. The T-Ruby design system. *Formal Methods in System Design*, 11(3): 239–264, 1997.

T. Sheard. Types and hardware description languages. In A. Martin, C. Seger, and M. Sheeran, editors, *Hardware Design and Functional Languages*, 2007.

T Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *Proceedings of the 2002 Haskell Workshop (Haskell 2002)*, pages 1–16, 2002. ACM, New York, NY.

M. Sheeran. μFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*, pages 104–112, 1984. ACM, New York, NY.

M. Sheeran. Describing and reasoning about circuits using relations. In McEvoy and Tucker (1990b), chapter 6.

M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computing Science*, 11(7):1135–1158, 2005.

M. Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *Journal of Functional Programming*, 21(1):59–114, 2011.

N. V. Shilov and N. O. Garanina. Model checking knowledge and fixpoints. In Zoltán Ésik and Anna Ingólfsdóttir, editors, *Fixed Points in Computer Science (FICS '02)*, volume NS-02-2 of *BRICS Notes Series*, pages 25–39, 2002. University of Århus, Århus, Denmark.

N. V. Shilov and N. O. Garanina. Well-structured model checking of multiagent systems. In I. Virbitskaite and A. Voronkov, editors, *Ershov Memorial Conference*, volume 4378 of *LNCS*, pages 363–376. Springer, 2006.

N. V. Shilov, N. O. Garanina, and K.-M. Choe. Update and abstraction in model checking of knowledge and branching time. *Fundamenta Informaticae*, 72(1-3):347–361, 2006.

T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proceedings of the 1996 European Conference on Design and Test (EDTC '96)*, 1996. IEEE Computer Society, Washington, DC.

Y. Shoham, editor. *Proceedings of the 6th Conference on Theoretical Aspects of Rationality and Knowledge (TARK '96)*, 1996. Morgan Kaufmann.

Y. Shoham and K. Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations.* Cambridge University Press, New York, NY, USA, 2008.

A. K. Simpson and G. D. Plotkin. Complete axioms for categorical fixed-point operators. In *Proceedings of the 15th IEEE Symposium on Logic in Computer Science (LICS '00)*, pages 30–41, 2000. IEEE Computer Society, Washington, DC.

S. Singh. Designing reconfigurable systems in Lava. In *17th International Conference on VLSI Design (VLSI Design 2004)*, pages 299–306, 2004. IEEE Computer Society, Washington, DC.

S. Singh. New parallel programming techniques for hardware design. In *IFIP WG 10.5 International Conference on Very Large Scale Integration of System-on-Chip (IFIP VLSI-SoC '07)*, pages 163–167, 2007. IEEE, Los Alamitos, CA.

S. Singh. The RLOC is dead – long live the RLOC. In J. Wawrzynek and K. Compton, editors, *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays (FPGA '11)*, pages 185–188, 2011. ACM, New York, NY.

S. Singh and P. James-Roxby. Lava and JBits: From HDL to bitstream in seconds. In *Proceedings of the the 9th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, pages 91–100, 2001. IEEE Computer Society, Washington, DC.

A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

R. Smullyan. *The Lady or the Tiger?* Oxford University Press, 1982.

V. Stavridou. *Formal Methods in Circuit Design.* Number 37 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, U.K., 1993.

V. Stavridou. Gordon's computer: A hardware verification case study in OBJ3. *Formal Methods in System Design*, 4(3):265–310, 1994.

R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, 2004.

C. Sternagel and R. Thiemann. Executable transitive closures of finite relations. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sf.net/entries/Transitive-Closure.shtml, 2011. Formal proof development.

J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

K. Su, A. Sattar, and X. Luo. Model checking temporal logics of knowledge via OBDDs. *The Computer Journal*, 50(4):403–420, 2007.

P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proceedings of the 13th Annual Symposium on Computer Architecture (ISCA '86)*, pages 414–423, 1986.

Tim Sweeney. The end of the GPU roadmap, 2009. Keynote at High Performance Graphics.

S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS*, pages 184–207, 1996. Springer, Berlin.

P. F. Syverson. Knowledge, belief, and semantics in the analysis of cryptographic protocols. *Journal of Computer Security*, 1(3-4):317–334, 1992.

W. Taha. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trivial (extended abstract). In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00)*, pages 34–43, 2000. ACM, New York, NY.

M. Tullsen. *PATH, a Program Transformation System for Haskell*. PhD thesis, Yale University, New Haven, CT, USA, 2002. URL `http://www.cs.yale.edu/publications/techreports/tr1229.pdf`.

A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528, 1996. Springer, Berlin.

A. Valmari. Fast brief practical DFA minimization. *Information Processing Letters*, 112(6), 2012.

W. van der Hoek and M. Wooldridge. Model checking cooperation, knowledge, and time — a case study. *Research in Economics*, 57(3):235 – 265, 2003.

R. van der Meyden. Knowledge based programs: On the complexity of perfect recall in finite environments. In Shoham (1996), pages 31–49.

R. van der Meyden. Finite state implementations of knowledge-based programs. In V. Chandru and V. Vinay, editors, *Proceedings of the 16th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS '96)*, volume 1180 of *LNCS*, pages 262–273, 1996b. Springer, Berlin.

R. van der Meyden. Constructing finite state implementations of knowledge-based programs with perfect recall. In L. Cavedon, A. S. Rao, and W. Wobcke, editors, *PRICAI Workshop on Intelligent Agent Systems*, volume 1209 of *LNCS*, pages 135–151. Springer, 1996c.

R. van der Meyden. Common knowledge and update in finite environments. *Information and Computation*, 140(2):115–157, 1998.

R. van der Meyden and N. V. Shilov. Model checking knowledge and time in systems with perfect recall (extended abstract). In C. P. Rangan, V. Raman, and R. Ramanujam, editors, *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '99)*, volume 1738 of *LNCS*, pages 432–445, 1999. Springer, Berlin.

R. van der Meyden and K. Su. Symbolic model checking the knowledge of the dining cryptographers. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW '04)*, pages 280–, 2004. IEEE Computer Society, Washington, DC.

R. van der Meyden and M. Y. Vardi. Synthesis from knowledge-based specifications (extended abstract). In D. Sangiorgi and R. de Simone, editors, *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR '98)*, volume 1466 of *LNCS*, pages 34–49, 1998. Springer, Berlin.

R. van der Meyden and T. Wilke. Synthesis of distributed systems from knowledge-based specifications. In M. Abadi and L. de Alfaro, editors, *Proceedings of the 16th International*

Conference on Concurrency Theory (CONCUR '05), volume 3653 of *LNCS*, pages 562–576, 2005. Springer, Berlin.

R. van der Meyden and C. Zhang. Algorithmic verification of noninterference properties. *Electronic Notes on Theoretical Computer Science*, 168:61–75, 2007.

H. van Ditmarsch, W. van der Hoek, R. van der Meyden, and J. Ruan. Model checking Russian cards. *Electronic Notes on Theoretical Computer Science*, 149(2):105–123, 2006.

H. van Ditmarsch, J. Ruan, and R. Verbrugge. Sum and product in dynamic epistemic logic. *Journal of Logic and Computation*, 18(4):563–588, 2008.

H. van Ditmarsch, W. van der Hoek, and J. Ruan. Connecting dynamic epistemic and temporal epistemic logics. *Logic Journal of the IGPL*, 2011. To appear.

J. van Eijck. DEMO — a demo of epistemic modelling. In J. van Benthem, D. Gabbay, and B. Löwe, editors, *Interactive Logic — Proceedings of the 7th Augustus de Morgan Workshop*, volume 1 of *Texts in Logic and Games*. Amsterdam University Press, 2007.

J. van Eijck and M. Stokhof. The gamut of dynamic logics. In D. M. Gabbay and J. Woods, editors, *Logic and Modalities in the Twentieth Century*, volume 6 of *The Handbook of the History of Logic*, chapter 7. Elsevier, 2006.

M. Y. Vardi. Implementing knowledge-based programs. In Shoham (1996), pages 15–30.

M. Y. Vardi and P. Wolper. Automata theoretic techniques for modal logics of programs (extended abstract). In *Proceedings of the 16th ACM Symposium on Theory of Computing (STOC '84)*, pages 446–456, 1984. ACM, New York, NY.

S. Vasudevan, B. DeCleene, N. Immerman, J. F. Kurose, and D. F. Towsley. Leader election algorithms for wireless ad hoc networks. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX '03)*, pages 261–272, 2003. IEEE Computer Society, Washington, DC.

T. L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, 2004.

J. Vuillemin. On circuits and numbers. *IEEE Trans. Computers*, 43(8):868–879, 1994.

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. (Special issue of selected papers from the 6th ACM Conference on Lisp and Functional Programming (LFP '90).).

P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 60–76, 1989. ACM, New York, NY.

M. Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

M. Wand and D. Vaillancourt. Relating models of backtracking. In C. Okasaki and K. Fisher, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming (ICFP '04)*, pages 54–65, 2004. ACM, New York, NY.

G. Winskel. Lectures on models and logic of MOS circuits. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*. Springer, Berlin, 1986.

G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, MA, 1993.

P. Wolper. Algorithms for synthesizing reactive systems: A perspective (abstract). In P. Flener, editor, *Proceedings of the 8th International Workshop on Logic Programming Synthesis and Transformation (LOPSTR'98)*, volume 1559 of *LNCS*, page 308, 1998. Springer, Berlin.

B. Wozna, A. Lomuscio, and W. Penczek. Bounded model checking for knowledge and real time. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '05)*, 2005. ACM, New York, NY.