593 6[illegible]

NASA/CP-2002-211736

Theorem Proving in Higher Order Logics

Edited by
Victor A. Carreño
Langley Research Center, Hampton, Virginia

César A. Muñoz
Institute for Computer Applications in Science and Engineering
Langley Research Center, Hampton, Virginia

Sofiène Tahar
Concordia University, Montreal, Canada

August 2002

NASA/CP-2002-211736

Theorem Proving in Higher Order Logics

Edited by
Víctor A. Carreño
Langley Research Center, Hampton, Virginia

César A. Muñoz
Institute for Computer Applications in Science and Engineering
Langley Research Center, Hampton, Virginia

Sofiène Tahar
Concordia University, Montreal, Canada

Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2002, held in Hampton, Virginia
August 20-23, 2002
and
Proceedings of the workshop on Formalising Continuous Mathematics, FCM 2002, held in Hampton, Virginia,
August 19, 2002

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

August 2002

Preface

This volume contains the Track B (Work in Progress) proceedings of the *15th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs 2002, held from the 20th to 23rd of August 2002, and the proceedings of the workshop on *Formalising Continuous Mathematics*, FCM 2002, held on the 19th of August 2002. These events were collocated in Hampton, Virginia, USA. A volume containing the Track A papers of the conference has been published as Springer-Verlag's Lecture Notes in Computer Science Volume 2410.

The TPHOLs International Conference serves as a venue for the presentation of work in theorem proving in higher-order logics, and related areas in deduction, formal specification, software and hardware verification, and other applications. Fourteen papers were submitted to Track B, which are included in this volume. Authors of Track B papers gave short introductory talks that were followed by an open poster session.

The FCM 2002 Workshop aimed to bring together researchers working on the formalisation of continuous mathematics in theorem proving systems with those needing such libraries for their applications. Over the last few years there has been great interest in formalising real and complex analysis. Many of the major higher order theorem proving systems now have a formalisation of the real numbers and various levels of real analysis support. Some work has also been done on formalising complex analysis including standard and non-standard analysis. This work is of interest in a number of application areas, such as formal methods development for hardware and software application and computer supported mathematics. The FCM 2002 consisted of three papers, presented by their authors at the workshop venue, and one invited talk by John Harrison (Intel Corporation). The three papers were accepted for publication in this volume.

All 17 papers in this volume were reviewed for relevance and quality of presentation by at least one person appointed by the program committee of each event.

Organization

TPHOLs 2002 is organized by NASA Langley and ICASE in collaboration with Concordia University. FCM 2002 is organized by the University of Reading and ICASE in collaboration with Intel Corporation and INRIA.

TPHOLs 2002 Organizing Committee

Conference Chair:	Víctor A. Carreño (NASA Langley)
Program Chair:	César A. Muñoz (ICASE - NASA LaRC)
	Sofiène Tahar (Concordia University)

FCM 2002 Organizing Committee

Workshop Chair:	Andrew A. Adams (University of Reading)
Program Chair:	Hanne Gottliebsen (ICASE - NASA LaRC)
	John Harrison (Intel Corporation)
	Laurent Théry (INRIA)

Table of Contents

FCM 2002 Workshop Papers

A Weakly-Typed Higher Order Logic with General Lambda Terms and Y Combinator

James H. Andrews[1]

Department of Computer Science
University of Western Ontario
London, Ontario, Canada N6A 5B7
andrews@csd.uwo.ca

Abstract. We define a higher order logic which has only a weak notion of type, and which permits all terms of the untyped lambda calculus and allows the use of the Y combinator in writing recursive predicates. The consistency of the logic is maintained by a distinction between use and mention, as in Gilmore's logics. We give a consistent model theory and a proof system which is valid with respect to the model theory. We also give examples showing what formulas can and cannot be used in the logic.

1 Introduction

The type system of a new higher order logic must be designed with care. Whenever we try to make the logic more expressive by permitting more well-typed terms, we risk making the logic inconsistent; for instance, Church's higher order logic [Chu40] cannot consistently be extended to permit the rather modestly-extended terms of ML [Coq86]. However, greater expressivity allows us to make more concise, intuitive, and general descriptions of the concepts we want to describe.

Most higher order logics follow the pattern of Church's original: one or more types are assigned to every term, and the type system is enforced with each rule of inference of the logic. Recently, however, Gilmore and others [Gil97,AK96] have been exploring higher order logics with only a weak notion of type, and no type enforcement across all terms. Consistency is maintained, not by types, but by a rigorous distinction between "use" and "mention" of predicate variables. In such logics, all the terms of the untyped lambda-calculus are permitted, and lambda-application can be used at the level of formulas as well.

There is a price to be paid for the greater expressiveness in this area, of course; certain variables cannot be used in certain positions in axioms. We have reason to believe, however, that the restriction may not be important for many computer science applications. These logics give a very different view of higher order logic, which may be useful in places where traditional higher order logics are not able to go.

In this paper, we extend Gilmore's ideas on the logic NaDSyL [Gil97] by defining a logic which further weakens the type system, allowing the use of the Y combinator. This weakening allows general recursive predicates to be defined and passed as parameters to other recursive predicates. We also present the proof system in the form of a conventional sequent calculus.

Gilmore himself has recently moved in the direction of stronger types, producing a logic intermediate between NaDSyL and Church's type theory [Gil01,Gil02]. The most relevant other related work that we are aware of is that of Kamareddine, who defines a logic which gives a type to the Y combinator without permitting general lambda terms [Kam92]. Our model theory is also similar in many ways to that of Chen, Kifer and Warren's HiLog [CKW89].

In section 2, we give a syntax of terms and a semantics for the logic, and prove the semantics consistent. In section 3, we give a proof system, and prove it valid with respect to the semantics. In section 4, we illustrate the expressiveness of the logic, and the bounds on that expressiveness, by showing what forms of terms can and cannot be used in it; we also speculate about the consequent usefulness of the system in computer science and theorem-proving. In section 5, we give some conclusions. This is work in progress: we have not yet proven cut-elimination, although the proof system is designed to facilitate it; and we have not undertaken a thorough comparison of the recursive predicate constructs to those in other higher order logics.

2 Syntax and Semantics

Here we present the elementary syntax of the language, define simple term models for the logic, and prove some consistency and substitutivity results.

2.1 Elementary Syntax

There are two sorts of variables, *use* and *mention*. We assume sets $\mathcal{X}_u$ of use variables and $\mathcal{X}_m$ of mention variables; the set $\mathcal{X}$ of variables is $\mathcal{X}_u \cup \mathcal{X}_m$. We assume a set $\mathcal{P}$ of predicate names and $\mathcal{C}$ of constants. The syntax of terms $\mathcal{T}$ in BNF is:
$\mathcal{T} \;::=\; x \mid p \mid c \mid K \mid (\mathcal{T}\ \mathcal{T}) \mid \lambda x.\mathcal{T}$
where x is a variable; p is a predicate name; c is a constant; and K is one of the connectives *and*, *not*, and *forall*. We use x, y, z as metavariables standing for either use or mention variables, and X, Y, Z as metavariables standing for use variables in particular. We use p, q, r as metavariables standing for predicate names, and a, b, c as metavariables standing for constants. We use M, N as metavariables standing for arbitrary terms. All metavariables may be possibly primed or subscripted.

As is standard, we write the term $(\ldots((M\ N_1)N_2)\ldots N_n)$ as $(M\ N_1\ \ldots\ N_n)$. We write $(and\ M\ N)$, $(not\ M)$, and $(forall\ \lambda x.M)$ as $M \& N$, $\neg M$, and $\forall x.M$ respectively. We define the notions of *free variables* and *variable substitution* in the usual way. We define α- and *β-convertibility* in the usual way, treating connectives as if they were constants. Two terms are *$\alpha\beta$-equivalent* if they are convertible to the same term via an arbitrary number of α- or β-conversion steps.

The notions of "use term" and "mention term" are central to the semantics and proof theory. A *mention term* is a term with no free use variables. A *use term* is a use variable, a predicate name, or a use term applied to a mention term; in other words, a use term is one of the form $(M\ N_1\ \ldots\ N_n)$, where M is a use variable or a predicate name, and each N_i has no free use variables. The significance of use terms is that they are the only ones given an *a priori* assignment of truth value in the semantics. Examples of use terms include $(p\ a\ b)$, $(X\ a\ b)$, $(p\ q\ r)$, $(p\ x\ y)$, and $(X\ x\ p)$, where x, y are mention variables. Examples of terms which are *not* use terms are $(c\ a\ b)$, $(x\ a\ b)$, and $(p\ a\ X)$, where x is a mention variable.

We denote the sets of use terms and mention terms by $\mathcal{T}_u$ and $\mathcal{T}_m$, respectively, and the sets of *ground* use and mention terms by $\mathcal{G}_u$ and $\mathcal{G}_m$, respectively. We denote the set of all terms by $\mathcal{T}$. Terms which are neither use nor mention terms are still considered well-formed, and can appear in formal derivations.

2.2 Model Theory

To show consistency and to provide a reference point for the proof theory, we define simple term models. These models correspond to Gilmore's models for NaDSyL [Gil97] in the same way that standard models correspond to nonstandard, Henkin-style models [Hen50]: they do not require the model to select denotations for use variables from a single given set. This relaxation simplifies the semantics.

We use the symbols T, F to denote the truth values "true" and "false", respectively. A *model* consists of a total function v and a countably infinite sequence $v_0, v_1, v_2, \ldots$ of total functions, such that:

- $v : \mathcal{X}_m \mapsto \mathcal{G}_m$;
- If N and N' are $\alpha\beta$-equivalent mention terms, then $v(N) = v(N')$;
- $v_0 : (\mathcal{X}_u \cup \mathcal{P}) \mapsto \{T, F\}$;
- For every $i > 0$, $v_i : (\mathcal{X}_u \cup \mathcal{P}) \mapsto (\mathcal{G}_m^i \mapsto \{T, F\})$; and
- If N and N' are $\alpha\beta$-equivalent mention terms, then $v_i(M)(\ldots, N, \ldots)) = v_i(M)(\ldots, N', \ldots))$.

We now extend the function v to all mention terms. Let the extension v_S of v w.r.t. a set $S \subseteq \mathcal{X}$ of variables be defined as follows:

- $v_S(x) = x$ if $x \in S$;
- $v_S(x) = v(x)$ if $x \notin S$ but $x \in \mathcal{X}_m$;
- $v_S(M) = M$ if M is a predicate name, constant, or connective;
- $v_S(M\ N) = (v_S(M)\ v_S(N))$;
- $v_S(\lambda x.M) = \lambda x.(v_{(S \cup \{x\})}(M))$.

We then write $v(M)$ for $v_{\{\}}(M)$. Note that $v(M)$ is undefined for free use variables, or indeed for any term containing a free use variable.

Given a model $(v, v_0, v_1, v_2, \ldots)$, we also extend the functions v_i to all use terms. We define $v_i(M\ N_1\ \ldots\ N_j)(N'_1, \ldots, N'_i)$ to be $v_{(j+i)}(M)(v(N_1), \ldots, v(N_j), N'_1, \ldots, N'_i)$, for all $i \geq 0$, $j \geq 1$, where $M \in (\mathcal{X}_u \cup \mathcal{P})$, $N_k \in \mathcal{T}_m$ for all $1 \leq k \leq j$, and $N'_k \in \mathcal{G}_m$ for all $1 \leq k \leq i$.

Given a use or mention variable x, a model $\mathcal{M}' = (v', v'_0, v'_1, \ldots)$ is an *x-variant* of another model $\mathcal{M} = (v, v_0, v_1, \ldots)$ if, for all mention variables y not identical to x, $v'(y) = v(y)$, and for all use variables Y not identical to x, $v'_i(Y) = v_i(Y)$ for all $i \geq 0$.

A *signed term* is a term preceded by a + or − sign, denoting truth or falsehood. We say that a model $\mathcal{M} = (v, v_0, v_1, \ldots)$ *entails* a signed term $\pm M$ *at stage* i, in symbols $\mathcal{M} \models_i \pm M$, just in the following cases.

- $\mathcal{M} \models_0 +(M\ N_1\ N_2\ \ldots\ N_n)$ if $M \in (\mathcal{X}_u \cup \mathcal{P})$, and $v_n(M)(v(N_1), v(N_2), \ldots, v(N_n)) = T$.
- $\mathcal{M} \models_0 -(M\ N_1\ N_2\ \ldots\ N_n)$ if $M \in (\mathcal{X}_u \cup \mathcal{P})$, and $v_n(M)(v(N_1), v(N_2), \ldots, v(N_n)) = F$.
- $\mathcal{M} \models_{i+1} +(M\&N)$ if $\mathcal{M} \models_i +M$ and $\mathcal{M} \models_i +N$.
- $\mathcal{M} \models_{i+1} -(M\&N)$ if $\mathcal{M} \models_i -M$ or $\mathcal{M} \models_i -N$.
- $\mathcal{M} \models_{i+1} +(\neg M)$ if $\mathcal{M} \models_i -M$.
- $\mathcal{M} \models_{i+1} -(\neg M)$ if $\mathcal{M} \models_i +M$.
- $\mathcal{M} \models_{i+1} +(\forall x.M)$ if for every x-variant $\mathcal{M}'$ of $\mathcal{M}$, $\mathcal{M}' \models_i +M$.
- $\mathcal{M} \models_{i+1} -(\forall x.M)$ if for some x-variant $\mathcal{M}'$ of $\mathcal{M}$, $\mathcal{M}' \models_i -M$.
- $\mathcal{M} \models_{i+1} +((\lambda x.M)\ N_1\ N_2\ \ldots N_n)$ if $\mathcal{M} \models_i +(M[x := N_1]\ N_2\ \ldots\ N_n)$.
- $\mathcal{M} \models_{i+1} -((\lambda x.M)\ N_1\ N_2\ \ldots N_n)$ if $\mathcal{M} \models_i -(M[x := N_1]\ N_2\ \ldots\ N_n)$.

We say that $\mathcal{M}$ *entails* $\pm M$, in symbols $\mathcal{M} \models \pm M$, if there is some $i \geq 0$ such that $\mathcal{M} \models_i \pm M$.

As an example, consider a model $\mathcal{M}$ and a use variable X. By the definition of model, either $\mathcal{M} \models -X$ or $\mathcal{M} \models +X$. Therefore either $\mathcal{M} \models -X$ or $\mathcal{M} \models -(\neg X)$, so $\mathcal{M} \models -(X\&(\neg X))$, and thus $\mathcal{M} \models +(\neg(X\&(\neg X)))$. Because this is true for every $\mathcal{M}$, it is also the case that $\mathcal{M} \models +\forall X(\neg(X\&(\neg X)))$. Expressing this formula more conventionally, $\mathcal{M} \models +\forall X(X \Rightarrow X)$.

An important point to note is the condition on the clause for $-(M\&N)$ above. $-(M\&N)$ is entailed if *either* $-M$ or $-N$ is entailed: the other conjunct does not have to be assigned a truth value at all by the model. This is in contrast to Gilmore's semantics for NaDSyL [Gil97], in which the other conjunct had to be assigned a truth value. The relaxation of the restriction makes the Y-combinator more useful. We can never entirely eliminate the Y-combinator from a typical recursive predicate term by beta-conversion, and thus can never convert a term with a Y-combinator into one in which all atomic terms are use terms. However, the $-(M\&N)$ clause allows, for instance, $(M\&N)$ to be false when M is false, regardless of whether N contains an irreducible term such as a Y-combinator.

2.3 Consistency and Substitutivity

Here we prove that it is impossible for a term to be both true and false in a model. We also prove some results concerning the substitution of mention terms for mention variables and use terms for use variables. These substitutivity results will be useful in later proving soundness of the proof system.

Theorem 1. *For every model $\mathcal{M}$ and every term M, it is not the case that both $\mathcal{M} \models +M$ and $\mathcal{M} \models -M$.*

Proof. If $\mathcal{M} \models \pm M$, then for some $i \geq 0$, $\mathcal{M} \models_i \pm M$. The proof is by induction on i. If $i = 0$, then M must be a use term, and by definition the model must assign one unique truth value to the term. Otherwise ($i > 0$), cases are on the form of M. If M is $(M_1\&M_2)$, then $\mathcal{M} \models_i +M$ iff $\mathcal{M} \models_{i-1} +M_1$ and $\mathcal{M} \models_{i-1} +M_2$. But in this case, by the IH, we can have neither $\mathcal{M} \models_{i-1} -M_1$ nor $\mathcal{M} \models_{i-1} -M_2$, and therefore we cannot have $\mathcal{M} \models_i -M$. Conversely, if $\mathcal{M} \models_i -M$, then either $\mathcal{M} \models_{i-1} -M_1$ or $\mathcal{M} \models_{i-1} -M_2$. But in neither case can we have both $\mathcal{M} \models_{i-1} +M_1$ and $\mathcal{M} \models_{i-1} +M_2$, and therefore we cannot have $\mathcal{M} \models_i +M$. The other cases follow the same straightforward reasoning. □

We define the *complexity* of a term M, in symbols $k(M)$, as follows. $k(\lambda x.M) = k(M) + 1$; $k(M\ N) = max(k(M), k(N)) + 1$; and $k(M) = 0$ when M is a variable, constant, predicate name or connective.

Theorem 2. *Let x be a mention variable and M a mention term. For every mention term N and every model $\mathcal{M}$ such that $v(x) = v(M)$, $v(N) = v(N[x := M])$.*

Proof. By induction on the complexity of N. Cases are straightforward. □

Theorem 3. *Let x be a mention variable and M a mention term. For every $i \geq 0$, every term N and every model $\mathcal{M}$ such that $v(x) = v(M)$, $\mathcal{M} \models_i \pm N$ iff $\mathcal{M} \models_i \pm N[x := M]$.*

Proof. By induction on i.

If $i = 0$, then N is a use term, of the form $(N_0\ N_1\ N_2\ \ldots\ N_n)$, where N_0 is a use variable or predicate name. By the definition of $\models$, $\mathcal{M} \models_0 +(N_0\ N_1\ N_2\ \ldots\ N_n)$ iff $v_n(N_0)(v(N_1), v(N_2), \ldots, v(N_n)) = T$. Because of the last theorem, this is true iff $v_n(N_0)(v(N_1[x := M]), v(N_2[x := M]), \ldots, v(N_n[x := M])) = T$. ($x$ does not appear free in N_0, so $N_0[x := M]$ is N_0.) By the definition of $\models$ again, the previous statement is true iff $\mathcal{M} \models_0 +(N_0\ N_1[x := M]\ N_2[x := M]\ \ldots\ N_n[x := M])$; that is, iff $\mathcal{M} \models_0 +(N_0\ N_1\ N_2\ \ldots\ N_n)[x := M]$. The subcase with $-$ instead of $+$ is similar.

If $i > 0$, then one of the other cases of the definition of the $\models_i$ relation holds. The cases of the propositional connectives are straightforward.

If N is $\forall y.N'$, then we assume without loss of generality that y is not x and that y does not appear in M. $\mathcal{M} \models_i +(\forall y.N')$ iff for every y-variant $\mathcal{M}'$ of $\mathcal{M}$, $\mathcal{M}' \models_{i-1} +N'$. By the induction hypothesis, this is true iff for every y-variant $\mathcal{M}'$ of $\mathcal{M}$, $\mathcal{M}' \models_{i-1} +N'[x := M]$. Because of the assumption, this is true iff $\mathcal{M} \models_i +(\forall y.N'[x := M])$; that is, iff $\mathcal{M} \models_i +(\forall y.N')[x := M]$. The subcase with $-$ instead of $+$ is similar.

If N is $((\lambda y.N_0)\ N_1\ N_2\ \ldots N_n)$, then $\mathcal{M} \models_i +N$ iff $\mathcal{M} \models_{i-1} +(N_0[y := N_1]\ N_2\ \ldots\ N_n)$. By the induction hypothesis, this is true iff $\mathcal{M} \models_{i-1} +((N_0[y := N_1])[x := M]\ N_2[x := M]\ \ldots\ N_n[x := M])$. We assume again that y is not x and that y does not appear in M. Therefore because of the properties of substitution, this is true iff $\mathcal{M} \models_{i-1} +((N_0[x := M])[y := (N_1[x := M])]\ N_2[x := M]\ \ldots\ N_n[x := M])$. By the definition of $\models$, this is true iff $\mathcal{M} \models_i +((\lambda y.N_0[x := M])\ N_1[x := M]\ N_2[x := M]\ \ldots\ N_n[x := M])$; that is, iff $\mathcal{M} \models_i +((\lambda y.N_0)\ N_1\ N_2\ \ldots\ N_n)[x := M]$. The subcase with $-$ instead of $+$ is similar. □

Theorem 4. *Let X be a use variable and M a use term. For every $i \geq 0$, every term N and every model $\mathcal{M}$ such that $v_j(X) = v_j(M)$ for every $j \geq 0$, $\mathcal{M} \models_i \pm N$ iff $\mathcal{M} \models \pm N[X := M]$.*

Proof. Similar to the proof of the last theorem, except for the base case of the induction.

If $i = 0$, then N is a use term, of the form $(N_0\ N_1\ N_2\ \ldots\ N_n)$, where N_0 is a use variable or predicate name and the N_is have no free use variables. If N_0 is not a use variable, or is a use variable different from X, then $N[X := M]$ is N, and the result follows immediately.

If N_0 is X, then $N[X := M]$ is $(M\ N_1\ N_2\ \ldots\ N_n)$. By the definition of $\models$, $\mathcal{M} \models_0 +(X\ N_1\ N_2\ \ldots\ N_n)$ iff $v_n(X)(v(N_1), v(N_2), \ldots, v(N_n))) = T$. By assumption, it is also the case that $v_n(M)(v(N_1), v(N_2), \ldots, v(N_n))) = T$. If M is a use variable or predicate, then the result follows immediately. Otherwise, M is $(M_0\ M_1\ \ldots\ M_m)$. By the definition of $v_n(M)$,

$$v_n(M)(v(N_1), v(N_2), \ldots, v(N_n))) = T = v_{m+n}(M_0)(v(M_1), \ldots, v(M_m), v(N_1), \ldots, v(N_n)))$$

Therefore $\mathcal{M} \models_0 +(M_0\ M_1\ \ldots\ M_m\ N_1\ \ldots\ N_n)$. Now it is also the case that

$$(M_0\ M_1\ \ldots\ M_m\ N_1\ \ldots\ N_n) = ((M_0\ M_1\ \ldots\ M_m)\ N_1\ \ldots\ N_n) = (M\ N_1\ \ldots\ N_n) = (X[X := M]\ N_1\ \ldots\ N_n)$$

Since X does not appear free in the N_is, this last term is equivalent to $(X\ N_1\ \ldots\ N_n)[X := M]$, or $N[X := M]$. Therefore $\mathcal{M} \models_0 +N[X := M]$, as desired. The subcase with $-$ instead of $+$ is similar. □

3 Proof Theory

Here we present a proof system for the logic, in the form of a classical sequent calculus similar to Gentzen's LK. We give notational preliminaries in section 3.1. In section 3.2, we prove soundness of the proof system with respect to the model theory; in section 3.3, we briefly discuss the prospects for cut elimination.

Reflexive entailment: $\dfrac{}{\Gamma, M \vdash \Delta, M'}$

where M, M' are use terms and $M =_{\alpha\beta} M'$

Thin/l: $\dfrac{\Gamma \vdash \Delta}{\Gamma, M \vdash \Delta}$ Thin/r: $\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, M}$

Con/l: $\dfrac{\Gamma, M, M \vdash \Delta}{\Gamma, M \vdash \Delta}$ Con/r: $\dfrac{\Gamma \vdash \Delta, M, M}{\Gamma \vdash \Delta, M}$

&/l: $\dfrac{\Gamma, M, N \vdash \Delta}{\Gamma, M \& N \vdash \Delta}$ &/r: $\dfrac{\Gamma \vdash \Delta, M \quad \Gamma \vdash \Delta, N}{\Gamma \vdash \Delta, M \& N}$

$\neg$/l: $\dfrac{\Gamma \vdash \Delta, M}{\Gamma, \neg M \vdash \Delta}$ $\neg$/r: $\dfrac{\Gamma, M \vdash \Delta}{\Gamma \vdash \Delta, \neg M}$

$\forall$/l: $\dfrac{\Gamma, M[x := N] \vdash \Delta}{\Gamma, \forall x.M \vdash \Delta}$ $\forall$/r: $\dfrac{\Gamma \vdash \Delta, M[x := y]}{\Gamma \vdash \Delta, \forall x.M}$

where x is a use (mention) variable, and N is a use (mention) term

where x is a use (mention) variable, and y is a use (mention) variable not appearing free in Γ, Δ, M

β/l: $\dfrac{\Gamma, (M[x := N_1]\ N_2\ \ldots\ N_n) \vdash \Delta}{\Gamma, ((\lambda x.M)\ N_1\ N_2\ \ldots\ N_n) \vdash \Delta}$ β/r: $\dfrac{\Gamma \vdash (M[x := N_1]\ N_2\ \ldots\ N_n), \Delta}{\Gamma \vdash ((\lambda x.M)\ N_1\ N_2\ \ldots\ N_n), \Delta}$

Cut: $\dfrac{\Gamma \vdash \Delta, M \quad M, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}$

Fig. 1. Proof rules for proof system G.

3.1 Notation

A *sequent* is an ordered pair of finite multisets of terms, written in the form $\Gamma \vdash \Delta$. We will use Γ and Δ as metavariables standing for multisets of terms; the notation Γ, Δ will mean $\Gamma \uplus \Delta$, and Γ, M will mean $\Gamma \uplus \{M\}$, where $\uplus$ is multiset union. We present the proof system in the form of Gentzen-style proof rules for sequents in Figure 1.

In Gilmore's original presentation [Gil97], he defines a semi-decidable set of *formulas*; the restriction on the $\forall$/l rule then essentially ensures that the term in the upper sequent must be a formula. The condition here is weaker, but simplifies the presentation.

Since the rules define a complete set of classical connectives, it is clear that we can introduce all the other classical connectives, such as $\vee$ (disjunction), $\rightarrow$ (implication) and $\exists$ (existential quantifier). We will use these connectives in the sequel without further comment.

3.2 Soundness

We say that a sequent $\Gamma \vdash \Delta$ is *valid* in a model $\mathcal{M}$ if either there is an $M \in \Gamma$ such that $\mathcal{M} \models -M$, or there is an $N \in \Delta$ such that $\mathcal{M} \models +N$. We say that a sequent $\Gamma \vdash \Delta$ is *valid* if it is valid in all models.

Theorem 5. *Every provable sequent is valid.*

Proof. By structural induction on the derivation. Cases are on the last rule applied. In the following, we write $\mathcal{M} \not\models -\Gamma$ to mean that for all $M \in \Gamma$, $\mathcal{M} \not\models -M$.

- Reflexive entailment: let the sequent be $\Gamma, M \vdash \Delta, M'$ such that $M =_{\alpha\beta} M'$ and M is a use term. Assume that $\mathcal{M} \not\models -\Gamma$ and $\mathcal{M} \not\models -M$. But since M is a use term, either $\mathcal{M} \models -M$ or $\mathcal{M} \models +M$; hence, $\mathcal{M} \models +M$. By the definition of model, $\mathcal{M} \models +M'$ as well.
- Thin/l: let the lower sequent be $\Gamma, M \vdash \Delta$. Assume that $\mathcal{M} \not\models -\Gamma$ and $\mathcal{M} \not\models -M$. Then by the induction hypothesis (IH) on the upper sequent, $\mathcal{M} \models +N$ for some $N \in \Delta$.
- Thin/r: Assume that $\mathcal{M} \not\models -\Gamma$. Then by the IH on the upper sequent, $\mathcal{M} \models +N$ for some $N \in \Delta$.
- Con/l, Con/r: straightforward.
- &/l: let the lower sequent be $\Gamma, M\&N \vdash \Delta$. Assume that $\mathcal{M} \not\models -\Gamma$ and $\mathcal{M} \not\models -(M\&N)$. Then neither $\mathcal{M} \models -M$ nor $\mathcal{M} \models -N$, because otherwise $\mathcal{M} \models -(M\&N)$. Thus, by the IH on the upper sequent, for some $N' \in \Delta$, $\mathcal{M} \models +N'$.
- &/r: let the lower sequent be $\Gamma \vdash \Delta, M\&N$. Assume that $\mathcal{M} \not\models -\Gamma$. There are two subcases. (a) If $\mathcal{M} \models +N'$ for some $N' \in \Delta$, then the result follows immediately. (b) Otherwise, by the IH on the upper left sequent, it must be the case that $\mathcal{M} \models +M$; and by the IH on the upper right sequent, it must be the case that $\mathcal{M} \models +N$. Thus, $\mathcal{M} \models +(M\&N)$.
- $\neg$/l: let the lower sequent be $\Gamma, \neg M \vdash \Delta$. Assume that $\mathcal{M} \not\models -\Gamma$ and $\mathcal{M} \not\models -(\neg M)$. Then $\mathcal{M} \not\models +M$, because otherwise $\mathcal{M} \models -(\neg M)$. Therefore by the IH on the upper sequent, there must be some $N \in \Delta$ such that $\mathcal{M} \models +N$.
- $\neg$/r: let the lower sequent be $\Gamma \vdash \Delta, \neg M$. Assume that $\mathcal{M} \not\models -\Gamma$. There are two subcases. (a) If there is some $N \in \Delta$ such that $\mathcal{M} \models +N$, then the result follows immediately. (b) Otherwise, by the IH on the upper sequent, $\mathcal{M} \models -M$; therefore $\mathcal{M} \models +(\neg M)$.
- $\forall$/l: let the lower sequent be $\Gamma, \forall x.M \vdash \Delta$. Assume without loss of generality that x does not appear free in Γ, Δ, N. Assume that $\mathcal{M} \not\models -\Gamma$ and $\mathcal{M} \not\models -(\forall x.M)$. Let $\mathcal{M} = (v, v_0, v_1, \ldots)$. We know that for any x-variant $\mathcal{M}'$ of $\mathcal{M}$, $\mathcal{M}' \not\models -M$, because otherwise $\mathcal{M} \models -(\forall x.M)$. There are two subcases. (a) If x is a mention variable, this means in particular that $\mathcal{M}' \not\models -M$ where $\mathcal{M}' = (v', v_0', v_1', \ldots)$ is an x-variant of $\mathcal{M}$ such that $v'(x)$ is the same as $v'(N)$. Therefore by Theorem 3, $\mathcal{M}' \not\models -M[x := N]$, and by the induction hypothesis, there must be some $N' \in \Delta$ such that $\mathcal{M}' \models +N'$. Since x does not appear free in N, it must be the case that $\mathcal{M} \models +N'$ as well. (b) Similarly, if x is a use variable, this means that $\mathcal{M}' \not\models -M$ where $\mathcal{M}' = (v', v_0', v_1', \ldots)$ is such that $v_i'(x)$ is the same as $v_i'(N)$ for all $i \geq 0$. The proof proceeds similarly, using Theorem 4.
- $\forall$/r: let the lower sequent be $\Gamma \vdash \Delta, \forall x.M$. Assume that $\mathcal{M} \not\models -\Gamma$. There are two subcases. (a) If there is some $N \in \Delta$ such that $\mathcal{M} \models +N$, then the result follows immediately. (b) Otherwise, by the IH on the upper sequent, $\mathcal{M} \models +M[x := y]$. But then for the x-variant $\mathcal{M}'$ of $\mathcal{M}$ in which $v(x) = v(y)$ (if x is a mention variable), or $v_i(x) = v_i(y)$ for all $i \geq 0$ (if x is a use variable), we have that $\mathcal{M}' \models +M$. Therefore $\mathcal{M} \models +(\forall x.M)$.

$$\begin{array}{c}
\hline
(X\ (\lambda x(\neg(x = x)))) \vdash (X\ (\lambda x(\neg(x = x)))) \\
\hline
\vdash (X\ (\lambda x(\neg(x = x)))) \Rightarrow (X\ (\lambda x(\neg(x = x)))) \\
\hline
\vdash \forall X.((X\ (\lambda x(\neg(x = x)))) \Rightarrow (X\ (\lambda x(\neg(x = x))))) \\
\hline
\hline
\vdash ((\lambda x(\neg(x = x))) = (\lambda x(\neg(x = x)))) \\
\hline
\neg((\lambda x(\neg(x = x))) = (\lambda x(\neg(x = x)))) \vdash \\
\hline
((\lambda x(\neg(x = x)))\ (\lambda x(\neg(x = x)))) \vdash \\
\hline
\vdash \neg((\lambda x(\neg(x = x)))\ (\lambda x(\neg(x = x)))) \\
\hline
\vdash (\lambda x(\neg(x\ x)))\ (\lambda x(\neg(x = x)))
\end{array}$$

Fig. 2. The proof that the empty set is in the Russell set.

- β/l, β/r: Straightforward applications of the induction hypothesis.
- Cut: let the lower sequent be $\Gamma \vdash \Delta$, and the cut term be M. Assume that $\mathcal{M} \not\models -\Gamma$. There are two subcases. (a) If there is some $N \in \Delta$ such that $\mathcal{M} \models +N$, then the result follows immediately. (b) Otherwise, from the IH on the upper left sequent, $\mathcal{M} \models +M$; hence $\mathcal{M} \not\models -M$. Therefore by the IH on the upper right sequent, there must be some $N \in \Delta$ such that $\mathcal{M} \models +N$. □

3.3 Cut Elimination

Because the proof system is valid with respect to the model theory, it inherits the model theory's consistency. Despite this, we would also like to prove cut-elimination. This would demonstrate that the proof system is compositional in a useful way. We have not completed the proof, but the proof system has been designed to allow it to proceed. For instance, any use term should be able to be substituted for a free use variable in a valid derivation, facilitating the important subcase of cut-elimination where the cut term is a universal quantification on a use variable.

4 Expressivity

Here we illustrate the expressivity of the logic, and the limits of that expressivity. In Section 4.1, we define and prove theorems about the Russell set. In Section 4.2, we show that the Y combinator can be defined and used. In Section 4.3, we show that Gilmore's operators for constructing recursive sets can still be used, and still have the effect of giving us proofs by induction without having to define an explicit induction rule. In Section 4.4, we show formulas that cannot be used in a truly useful way, and speculate on what consequences this has for the logic. Finally, in Section 4.5, we discuss "nonsense formulas" that can arise given the syntax and proof theory of the logic, and propose a familiar solution.

In this discussion, we will take the notation $M = N$ as shorthand for Frege's "identity of indiscernables" property, $\forall X.((X\ M) \Rightarrow (X\ N))$.

4.1 The Russell Set

The Russell set $\mathbf{R}$ can be expressed as $\lambda x(\neg(x\ x))$, and stands as usual for the set of all sets which are not members of themselves. Given that the empty set $\mathbf{E} \overset{\text{def}}{=} \lambda x(\neg(x = x))$ is not a member of itself, but the set of all terms $\mathbf{A} \overset{\text{def}}{=} \lambda x(x = x)$ is a member of itself, we should be able to prove both $(\mathbf{R}\ \mathbf{E})$ and $\neg(\mathbf{R}\ \mathbf{A})$.

Figures 2 and 3 show these derivations. (In all derivations in this paper, we occasionally insert "steps" in which we simply expand definitions.) These derivations demonstrate that the logic of this paper retains the expressiveness of Gilmore's NaDSyL [Gil97], which can also express these derivations. It also demonstrates that the logic is in this particular way more expressive than Church's [Chu40] and Kamareddine's [Kam92], in which the Russell set cannot be assigned a type and thus cannot appear in derivations.

The Russell set is not particularly useful by itself, but its presence is an indication of the difference between Gilmore logics and traditional type theories.

$$\begin{array}{c}
\hline
(X\ (\lambda x(x = x))) \vdash (X\ (\lambda x(x = x))) \\
\hline
\vdash ((X\ (\lambda x(x = x))) \Rightarrow (X\ (\lambda x(x = x)))) \\
\hline
\vdash \forall X.((X\ (\lambda x(x = x))) \Rightarrow (X\ (\lambda x(x = x)))) \\
\hline
\vdash ((\lambda x(x = x)) = (\lambda x(x = x))) \\
\hline
\vdash ((\lambda x(x = x))\ (\lambda x(x = x))) \\
\hline
\neg((\lambda x(x = x))\ (\lambda x(x = x))) \vdash \\
\hline
((\lambda x(\neg(x\ x)))\ (\lambda x(x = x))) \vdash \\
\hline
\vdash \neg((\lambda x(\neg(x\ x)))\ (\lambda x(x = x)))
\end{array}$$

Fig. 3. The proof that the set of all terms is not in the Russell set.

$$\begin{array}{c}
\vdash (s\ 0) = (s\ 0) \qquad \vdash ((\mathbf{Y}\ \mathbf{N})\ 0) \\
\hline
\vdash ((s\ 0) = (s\ 0)\&((\mathbf{Y}\ \mathbf{N})\ 0))) \\
\hline
\vdash \exists y((s\ 0) = (s\ y)\&((\mathbf{Y}\ \mathbf{N})\ y))) \\
\hline
\vdash (s\ 0) = 0, \exists y((s\ 0) = (s\ y)\&((\mathbf{Y}\ \mathbf{N})\ y))) \\
\hline
\vdash ((s\ 0) = 0 \vee \exists y((s\ 0) = (s\ y)\&((\mathbf{Y}\ \mathbf{N})\ y))) \\
\hline
\vdash ((\lambda x.(x = 0 \vee \exists y(x = (s\ y)\&((\mathbf{Y}\ \mathbf{N})\ y))))\ (s\ 0))
\end{array}$$

Fig. 4. Part of the proof that $(s\ 0)$ is an integer.

4.2 The Y Combinator

Let $\mathbf{V}(w)$ be $\lambda v.(w\ (v\ v))$. Then Curry's Y combinator can be defined as $\mathbf{Y} \overset{\text{def}}{=} \lambda w.(\mathbf{V}(w)\ \mathbf{V}(w))$. We define the set of integers as consisting of 0 and all terms of the form $(s\ n)$, where s and 0 are here assumed to be constants and n is an integer. If we define $\mathbf{N} \overset{\text{def}}{=} \lambda u.\lambda x.(x = 0 \vee \exists y(x = (s\ y)\&(u\ y)))$, then $(\mathbf{Y}\ \mathbf{N})$ expresses the set of integers.
Applying beta-reduction and definition expansion/contraction to this term yields the following.

$$\begin{aligned}
(\mathbf{Y}\ \mathbf{N}) &= (\lambda w.(\mathbf{V}(w)\ \mathbf{V}(w)))\ \mathbf{N} \\
&= (\mathbf{V}(\mathbf{N})\ \mathbf{V}(\mathbf{N})) \\
&= (\lambda v.(\mathbf{N}\ (v\ v)))\ \mathbf{V}(\mathbf{N})) \\
&= (\mathbf{N}\ (\mathbf{V}(\mathbf{N})\ \mathbf{V}(\mathbf{N}))) \\
&= (\mathbf{N}\ (\mathbf{Y}\ \mathbf{N})) \\
&= (\lambda u.\lambda x.(x = 0 \vee \exists y(x = (s\ y)\&(u\ y)))\ (\mathbf{Y}\ \mathbf{N})) \\
&= \lambda x.(x = 0 \vee \exists y(x = (s\ y)\&((\mathbf{Y}\ \mathbf{N})\ y)))
\end{aligned}$$

Figure 4 shows part of a derivation of the term $((\mathbf{Y}\ \mathbf{N})\ (s\ 0))$; this may be read as the statement that the term $(s\ 0)$ is an integer. The omitted parts of the derivation, at the top, are straightforward given this derivation and the previous derivations. This derivation demonstrates that the Y combinator can be used in this logic, as in Kamareddine's [Kam92]. The Y combinator is disallowed in Church's T system [Chu40] and in Gilmore's ITT [Gil01,Gil02] because it cannot be assigned a type; it is disallowed in NaDSyL [Gil97] because no term containing it could be considered a "formula".

Despite the presence of the Y combinator and implication, Curry's paradox [HS86] is avoided because the sequent $M \vdash M$ is not an axiom if M is an application of a lambda-abstraction. An attempt to prove Curry's paradoxical formula leads only to an infinite regress.

4.3 Gilmore's Operators

Gilmore [Gil97] defines a kind of "intersection" operator which can be used to construct recursively-defined sets. The intersection operator $\mathbf{L}$ is $\lambda X.\lambda x.\forall Y((X\ Y) \Rightarrow (Y\ x))$; given a property M and a term N, the term $(\mathbf{L}\ M\ N)$ will be true if N is a member of all sets with the property M. Let $\mathbf{Z}$ be the property of being a zero-successor set, i.e. the property of a set containing 0 and all of its successors. $\mathbf{Z}$ can be defined as $\lambda Z.((Z\ 0)\&\forall y((Z\ y) \Rightarrow (Z\ (s\ y))))$. Then the term $\mathbf{N}' \overset{\text{def}}{=} (\mathbf{L}\ \mathbf{Z})$ expresses the property that an individual is in every zero-successor set; that is, $\mathbf{N}'$ stands for the set of natural numbers.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(M\ x) \vdash (M\ x) \qquad \dfrac{\dfrac{\vdash (M\ 0) \qquad \dfrac{\dfrac{(M\ y) \vdash (M\ (s\ y))}{\vdash ((M\ y) \Rightarrow (M\ (s\ y)))}}{\vdash \forall y((M\ y) \Rightarrow (M\ (s\ y)))}}{\vdash ((M\ 0)\&\forall y((M\ y) \Rightarrow (M\ (s\ y))))}}{\vdash (M\ x), ((M\ 0)\&\forall y((M\ y) \Rightarrow (M\ (s\ y))))}}{(((M\ 0)\&\forall y((M\ y) \Rightarrow (M\ (s\ y)))) \Rightarrow (M\ x)) \vdash (M\ x)}}{\forall Y(((Y\ 0)\&\forall y((Y\ y) \Rightarrow (Y\ (s\ y)))) \Rightarrow (Y\ x)) \vdash (M\ x)}}{(\mathbf{N}'\ x) \vdash (M\ x)}}{\vdash (\mathbf{N}'\ x) \Rightarrow (M\ x)}}{\vdash \forall x((\mathbf{N}'\ x) \Rightarrow (M\ x))}$$

Fig. 5. Part of a proof that the use term M expresses a property held by all integers.

Applying equivalences and definition expansions yields the following equivalent definition for $\mathbf{N}'$:

$$\begin{aligned}(\mathbf{L}\ \mathbf{Z}) &= ((\lambda X.\lambda x.\forall Y((X\ Y) \Rightarrow (Y\ x)))\ \mathbf{Z})\\ &= \lambda x.\forall Y((\mathbf{Z}\ Y) \Rightarrow (Y\ x))\\ &= \lambda x.\forall Y(((\lambda Z.((Z\ 0)\&\forall y((Z\ y) \Rightarrow (Z\ (s\ y)))))\ Y) \Rightarrow (Y\ x)))\\ &= \lambda x.\forall Y(((Y\ 0)\&\forall y((Y\ y) \Rightarrow (Y\ (s\ y)))) \Rightarrow (Y\ x))\end{aligned}$$

The significance of this expression, as in Gilmore's previous logics, is that a proof of a property of all natural numbers automatically consists of a proof by induction. Figure 5 shows a generalized proof of the term schema $\forall x((\mathbf{N}'\ x) \Rightarrow (M\ x))$, that is "property M holds of all integers", for any given use term M. The derivation follows the general pattern of reducing the problem of proving the term to the problem of proving $(M\ 0)$ and $\forall y((M\ y) \Rightarrow (M\ (s\ y)))$.

This latter property may be useful in automated theorem proving, since it allows us to prove properties concerning recursively defined sets by induction, without explicitly requiring induction rules or tactics. Naturally, other tactics may be required, and it remains to be seen whether they are easier or harder to use.

4.4 Disallowed Formulas

Terms such as $(X\ Y)$, where X and Y are both free use variables, cannot appear in axioms. As a consequence of this, they are not really useful in any derivation except when one or both are bound by a lambda, although they are not excluded by any type restriction. The logic of this paper shares this property with Gilmore's two recent logics [Gil97,Gil01].

In contrast, of course, in most higher order logics from Church's [Chu40] to Kamareddine's [Kam92], a term of the form $(X\ Y)$ is allowed as long as the types are appropriate. This leads to many natural uses of predicate variables applied to others, such as that of assigning a value to a predicate variable and later applying it to another predicate. The fact that applying a variable to another comes so naturally for us may lead us to believe that we cannot do without it.

However, note that non-recursive and recursive properties of general predicates can still be defined in the logic of this paper. The property that a predicate is transitive, for instance, can be defined as $\lambda X.(\forall x \forall y \forall z.((X\ x\ y)\&(X\ y\ z) \Rightarrow (X\ x\ z)))$. This term can be applied to any predicate or use variable, or indeed to any term, including one using the Y combinator. The transitive closure operator on a binary relation can be defined using the Y combinator: if $\mathbf{T}$ is $\lambda Z \lambda X \lambda x \lambda y.((X\ x\ y) \vee \exists z((X\ x\ z)\&(Z\ X\ z\ y)))$, then $((\mathbf{Y}\ \mathbf{T})\ M)$ is the transitive closure of a term M. Again, M can be a predicate name, a use variable, or another term defined using the Y combinator. Applying $((\mathbf{Y}\ \mathbf{T})\ M)$ to two terms will result in a term which can have a derivation of the same general form as that in Figure 4.

This property gives us reason to believe that the logic of this paper can still be useful in working with higher order recursive definitions. It will always be impossible to use terms such as $\forall X \forall Y((X\ Y) \Rightarrow \ldots)$ as formulas, but such formulas may not be needed for some applications.

Moreover, note that a use variable can be applied to a particular predicate, such as in the term $(X\ p)$. This allows us to, for instance, state assumptions about the application of p to other terms on the left-hand side of a sequent, and assert properties of p itself on the right-hand side. We simply cannot generalize from a property about a given predicate p to a property of all use terms.

4.5 Nonsense Formulas

We have so far in this paper restricted our attention to formulas that "make sense" in an intuitive way. However, readers have probably noticed that there are many terms which can be written and can appear in formulas but don't make sense, such as (*and and*), $(\mathbf{Y}\ x)$, and $\forall x.(c)$. There are also many terms which make sense but cannot be construed as formulas, such as $\lambda x.(x = x)$; if we try to build a derivation of an ill-constructed formula, we may be faced with such a term in a formula position.

The natural way to address this problem is a familiar one: to impose a type system on the terms of the logic. Note, however, that since types are not needed for consistency, we can choose our type system based on considerations of expressivity. For instance, the Y combinator can be given a type in type systems with fixpoint types and universal types, such as those described by Cardelli [Car96]; such type systems are normally inaccessible to us in higher order logics *because* they allow such problematic terms. Using Gilmore-style logics, we can choose our type system to include such terms if we so desire.

The weak type system of use and mention variables used in the logic in this paper does not seem to interact with a more conventional type system except in one respect. Let us assume that o is the type of propositions. It does not seem to make sense to allow constants and mention variables to have a type ending in o (i.e., a type of the form $(A_1 \to A_2 \to A_3 \to \cdots \to o)$), since they can never appear as the head term of a formula in an axiom anyway. Conversely, it does not seem to make sense to allow predicate names and use variables to have any type not ending in o.

5 Conclusions and Future Work

We have shown another facet of the complex jewel of consistent higher-order logics, one close to that shown by Gilmore in his earlier work. Whether the logic of this paper turns out to be useful remains to be seen, but we believe that its existence should be interesting to the community of researchers working with higher order logic. Future work includes a proof of cut-elimination and more in-depth study of the nature of recursive and higher order definitions in comparison to previous logics.

6 Acknowledgments

Thanks especially to Paul Gilmore for many invaluable conversations about this work, and to Denby Wong for detailed discussions and important insights into soundness and cut-elimination. Thanks also to Peter Apostoli, Bharat Jayaraman, and George Tsiknis for important observations. This work was supported by Paul Gilmore's and the author's NSERC operating grant and by the FormalWare project, an initiative headed by Jeff Joyce and funded by the BC Advanced Systems Institute, Hughes Aircraft Canada Ltd., and Macdonald Dettwiler.

References

[AK96] Peter Apostoli and Akira Kanda. Parts of the continuum: towards a modern ontology of science. Accepted for publication, Poznan Studies in the Philosophy of Science and the Humanities, ed. Leszek Nowak, 1996.

[Car96] Luca Cardelli. Type systems. In *CRC Handbook of Computer Science and Engineering*, pages 2208–2236. CRC Press, 1996.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CKW89] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A first-order semantics of higher-order logic programming constructs. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, Ohio, October 1989.

[Coq86] Thierry Coquand. An analysis of Girard's paradox. In *First IEEE Symposium on Logic in Computer Science*, pages 227–236, Cambridge, Massachusetts, June 1986.

[Gil97] Paul C. Gilmore. NaDSyL and some applications. In *Proceedings of the Kurt Gödel Colloquium*, volume 1289 of *LNCS*, pages 153–166, Vienna, 1997. Springer.

[Gil01] Paul C. Gilmore. An intensional type theory: Motivation and cut-elimination. *Journal of Symbolic Logic*, 66(1):383–400, March 2001.

[Gil02] Paul C. Gilmore. *Logicism Renewed: Logical Foundations for Mathematics and Computer Science*. Association for Symbolic Logic, Poughkeepsie, NY, 2002. Forthcoming.

[Hen50] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81 91, 1950.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda Calculus*. Number 1 in London Mathematical Society Student Texts. Cambridge University Press, Cambridge, 1986.

[Kam92] Fairouz Kamareddine. A system at the cross-roads of functional and logic programming. *Science of Computer Programming*, 19:239 279, 1992.

Formal Design Environments

Brian Aydemir, Adam Granicz, and Jason Hickey*

California Institute of Technology, Pasadena CA 91125, USA
{emre,granicz,jyh}@cs.caltech.edu
http://mojave.cs.caltech.edu

Abstract. We present the design of a formal integrated design environment. The long-term goal of this effort is to allow seamless interaction between software production tools and formal design and analysis tools, especially between compilers and higher-order theorem provers. The work in this report is the initial design and architecture for integration of 1) the MetaPRL logical framework, 2) a multi-language compiler we call Mojave, and 3) a generic extensible parser we call Phobos. The integration is currently performed at the level of the Mojave functional intermediate representation, allowing the use of the theorem prover for program analysis, transformation, and optimization.

1 Introduction

We are developing formal integrated design environments (FIDEs) where formal and informal tools are used in a symbiotic relationship. That is, interactions between the formal and informal parts of the FIDE are bidirectional and interdependent.

Most, if not all, existing formal design environments do not allow bidirectional interactions, especially between the theorem prover and the compiler. Yet, the system would clearly benefit from closer interaction. For example, the compiler might be able to use the theorem prover for optimization, proof validation, or program transformation. The theorem prover would benefit from the ability to formalize its own code, especially tactics.

The larger need is for effective formal programming languages. By "effective" we mean that the languages should be general enough and efficient enough to use in software production. By "formal" we mean that programs can be both specified and verified. The compiler is responsible for efficiency, the prover for formality. In order to achieve both properties simultaneously, we argue that the theorem prover and compiler must interact closely (or, equivalently, one must be folded into the other).

In this paper, we describe our initial work integrating the MetaPRL logical framework with our Mojave multi-language compiler. There are several parts that are needed for integration: 1) the compiler and theorem prover must share a common language, 2) the compiler must allow for an extended program syntax that includes specification, and 3) the compiler and prover must also agree on a common program semantics, especially operational semantics. We present the following results:

- an architecture and implementation for the MetaPRL/Mojave formal design environment,
- a shared typed intermediate language, with semantics defined in the MetaPRL implementation of the NuPRL type theory,
- an extensible front-end, called Phobos, that uses the MetaPRL rewriting system for extending and defining programming languages,
- and examples of using the theorem prover for optimization.

Section 2 describes related work. Section 3 describes the MetaPRL, Mojave, and Phobos systems individually, and Section 4 presents the combined architecture. We give example applications in Section 5, and finish with a summary of future work.

* This work was supported by the ONR, grant N00014-01-1-0765; DARPA, grant F33615-98-C3613; and AFOSR, grant F49620-01-1-0361.

2 Related Work

This work initially started with the development of the MetaPRL system [6, 8]. MetaPRL is a logical framework, designed to allow *relations* between logics. MetaPRL is also designed as a "Logical Programming Environment" (LPE) where programs, type systems, proofs, and specifications can all be defined and related to one another.

One of the problems with the MetaPRL design is that it is a layered architecture. The theorem prover is layered above the OCaml compiler [16], and the connection is unidirectional. Any task (such as parsing, type inference, and scoping) that is assigned to the compiler is not available to the formal system, hindering effective formal software development.

In another related effort, we used the NuPRL system to optimize communication protocols for the Ensemble group communication system [12, 11]. Again, this project separated the prover from the compiler. To optimize a protocol, a parser would convert the protocol and requirements into an expression in the NuPRL type theory; the prover would apply optimization tactics to generate a "fast-path:" and the result would be printed as a ML file to be compiled by the OCaml compiler. While successful, this was awkward. Furthermore, optimization strategies were defined in NuPRL, not as part of the program code, making it difficult to synchronize the formal system with new Ensemble code releases. The architecture we propose in this paper is an effort to design a system where formal properties are "first-class" program properties, and the prover/compiler interaction is seamless.

In other related areas, Sannella and Tarlecki's Extended ML [9, 10] allows mixing of program implementation and formal specification for SML programs. The ACL2 system [3] allows extensive mixing of formal specification and Common Lisp programs. Nearly all other formal systems, including systems like HOL [5], PVS [4] and Isabelle [14], allow extensive reasoning about programs, but the prover is not coupled with a compiler as we are proposing in this paper.

3 Architectural Components

There are three major parts to our architecture. The MetaPRL system provides reasoning, the Mojave system provides compilation, and the Phobos system provides generic, extensible parsing. The overall system architecture is shown in Figure 1.

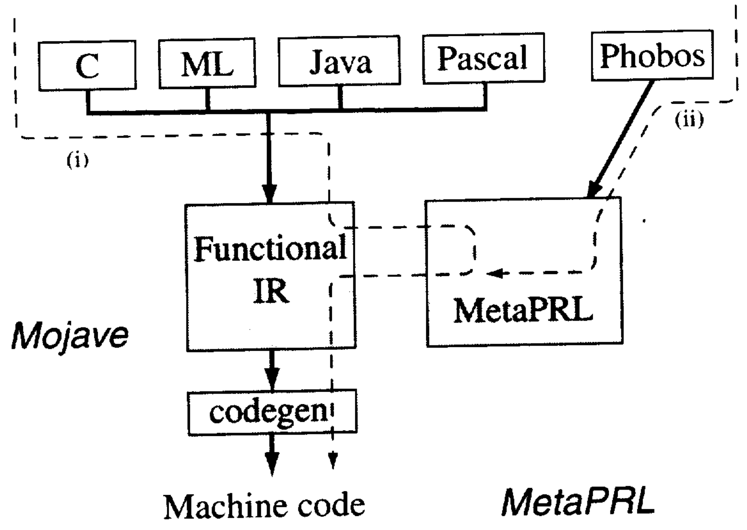


Fig. 1. System architecture: Path (i) corresponds to a traditional compilation path augmented by the theorem prover. Path (ii) uses the dynamically extensible front end.

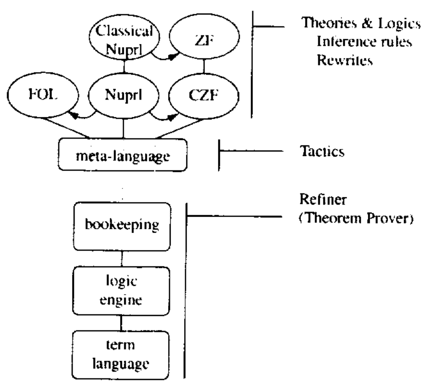


Fig. 2. MetaPRL system architecture

3.1 The MetaPRL system

MetaPRL is a *logical framework*, designed along the same architectural principles as the NuPRL system. The system architecture is shown in Figure 2. The refiner contains three parts: 1) a term module, which defines the system syntax, 2) a logic engine, for theorem proving, and 3) a bookeeping module to manage and validate proofs, and perform program extraction for constructive proofs.

The meta-language defines the language of *tactics* and *conversionals* (rewriting tactics), which are used to define decision procedures and proof heuristics. The entire MetaPRL system is written in OCaml, and OCaml is used as the language for tactics and conversionals.

The topmost layer in MetaPRL is the definition of theories and logics. A *theory* is defined by 1) its syntax (defined using terms), 2) its proof and rewriting rules, 3) its tactics and conversionals, 4) its theorems, expressed as derived inference and rewriting rules, and 5) other utilities for display and pretty-printing.

Judgments Inference rules are often, though not always, described in a sequent logic. For example, the following inference rule would describe the implies introduction rule in a propositional logic.

rule imp_intro H :
[main] $(H, v\colon A \vdash B) \longrightarrow$
[wf] $(H \vdash A\ \textit{type}) \longrightarrow$
$H \vdash A \Rightarrow B$

In this rule, the variables A, B, are meta-variables that represent arbitrary terms and H represents a context. The sequents labeled "main" and "wf" are the premises of the rule; "main" is the main premise, and "wf" is a well-formedness requirement. The $\longrightarrow$ operator is the meta-implication that MetaPRL uses to represent inference rules. The declaration of the `imp_intro` rule defines a tactic, called `imp_intro` that can be used to "refine" any goal of the form $H \vdash A \Rightarrow B$ into two subgoals. From here, it is straightforward, for example, to define a derived rule (a *theorem*) that would apply to sequents of the form $H \vdash A \Rightarrow B \Rightarrow C$. The proof would use the `imp_intro` rule twice, and there would be four premises.

Rewriting judgments are defined in a similar way. The rule for beta-reduction in the untyped lambda calculus would be expressed using the following rule.

rewrite beta : $\textit{apply}\{\textit{lambda}\{v.e_1[v]\}; e_2\} \longleftrightarrow e_1[e_2]$

This declaration defines a *conversion* called `beta` that can be applied within a proof to any redex, performing the substitution. Note that the statement of the rewrite uses second-order substitution [1, 13]. The pattern $e_1[v]$ represents a term in which the variable v is allowed to be free, and the term $e_1[e_2]$ represents e_1 with e_2 substituted for v.

Syntax, and terms All logical terms, including goals and subgoals, are expressed in the language of *terms*. The general syntax of all terms has three parts. Each term has 1) an operator-name (like "sum"), which is a unique name indicating the logic and component of a term; 2) a list of parameters representing constant values; and 3) a list of subterms with possible variable bindings. We use the following syntax to describe terms, based on the NuPRL definition [2]:

$$\underbrace{\mathit{opname}}_{\mathit{operator\ name}}\ \underbrace{[p_1;\cdots;p_n]}_{\mathit{parameters}}\underbrace{\{v_1.t_1;\cdots;v_m.t_m\}}_{\mathit{subterms}}$$

A few examples are shown at the right. Variables are terms with a string parameter for their name; numbers have an integer parameter. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b.

Displayed form	Term
1	`number[1]{}`
$\lambda x.b$	`lambda[]{x. b}`
$f(a)$	`apply[]{f; a}`
v	`variable["v"]{}`
$x+y$	`sum[]{x; y}`

3.2 The Mojave compiler

The Mojave multi-language compiler, shown in the left half of Figure 1, is made up of three major parts. The front-ends are responsible for compiling code to the functional intermediate representation (FIR), and the back-end is responsible for generating machine code from FIR. FIR type inference and optimizations form the middle stage of the compiler. The FIR is the primary concern for this paper; it is the language we are using for interaction with MetaPRL.

Functional Intermediate Representation The FIR is designed to be a minimal, but general-purpose typed intermediate language. The FIR has higher-order functions, polymorphism, and object types. We will describe it in two parts, first the type system, and then the programs.

The FIR type system The FIR type system is based on the polymorphic lambda calculus. The type system is shown in Figure 3. There are the normal scalars, including native integers with various bit precisions ($\mathbb{Z}_8,\ldots,\mathbb{Z}_{64}$) as well as "boxed" integers, enumerations $\{0\ldots i\}$ whose values range from 0 to $i-1$, and floating-point numbers. Enumerations are used to code several base types: the empty type *Void* is $\{0\ldots 0\}$, *Unit* is $\{0\ldots 1\}$, etc.

Functions have multiple arguments. The type $(t_1,\ldots,t_n)\to t$ is the space of functions taking arguments with types $t_1,\ldots,t_n$ and returning a value of type t.

Tuples $\langle t_1,\ldots,t_n\rangle$ are collections of elements having potentially different type. The t **array** represents variable-sized arrays of elements all having type t. The **data** type is used specifically for C: it represents a variable-sized data area containing elements of any type. Values of **data** type are not statically type-checked: it is not a type error to store an integer in a **data** area, and immediately fetch a value from the same location with some other type, but the runtime will raise an exception if the operation is unsafe.

Types are always defined relative to a context Γ that contains type definitions and scope information for polymorphic variables. The union type $\Lambda\alpha_1,\ldots,\alpha_n.\boldsymbol{t_1}+\cdots+\boldsymbol{t_m}$ is a polymorphic disjoint union of tuples $\boldsymbol{t_1},\ldots,\boldsymbol{t_m}$. A value with a disjoint union type is a tagged tuple of type $\boldsymbol{t_i}$ with tag i for some $i\in\{1,\ldots,n\}$. If v defines a union type $\Lambda\alpha_1,\ldots,\alpha_n.\boldsymbol{t_1}+\cdots+\boldsymbol{t_n}$, then the constructor type $\mathbf{const}(v[i],a_1,\ldots,a_n)$ denotes a tagged tuple of type $t_i[a_1,\ldots,a_n]$.

Polymorphism is expressed using the existential and universal types. A value of type $\exists\alpha.t$ has type t for some type α, and a value of type $\forall\alpha.t$ has type t for all types α.

An object type $\mathbf{Object}(v.t)$ is a recursive type definition denoting objects with description t.

FIR expressions The FIR expressions are in a mostly-functional intermediate form where the order of evaluation is explicit. *Atoms* are values that are either constants or variables, and the other expressions are computed over the atoms. Function definitions are stored in an environment Σ that also serves as a type assignment. The definitions are shown in Figure 4.

Expressions include explicit coercions and arithmetic as unary and binary operators.

Type	Description
$t ::= \textbf{boxed}(\mathbb{Z})$	Boxed Integers
$\mid \{0 \ldots i\}$	Integer enumerations
$\mid \mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}$	Native integers
$\mid \textbf{float}$	Floating-point numbers
$\mid (t_1, \ldots, t_n) \rightarrow t$	Function type
$\mid \textbf{const}(v[i], t_1, \ldots, t_n)$	Constructor type
$\mid \langle t_1, \ldots, t_n \rangle$	Tuple type
$\mid t\ \textbf{array}$	Array type
$\mid \textbf{data}$	Unsafe data
$\mid \alpha, \beta, \ldots$	Polymorphic type variables
$\mid v$	Type variables
$\mid v[t_1, \ldots, t_n]$	Type application
$\mid \exists \alpha.t$	Existential types
$\mid \forall \alpha.t$	Universal types
$\mid v.i$	Abstract type
$\mid \textbf{Object}(v.t)$	Object type
$d ::= \Lambda\alpha_1, \ldots, \alpha_n.\boldsymbol{t_1} + \cdots + \boldsymbol{t_m}$	Type abstraction
$\gamma ::= v = d \mid \alpha$	
$\Gamma ::= \gamma_1, \ldots, \gamma_n$	Type contexts

Fig. 3. The FIR type system

The "ext" call represents a call to the runtime, providing a method to issue system calls. Type definitions for system calls are provided as part of the compiler configuration, to ensure type safety. The tailcall provides the only other means for calling a function.

The "match" construction allows pattern matching against a constant set of numbers, represented as a list of intervals. Each match case defines a set s and an expression e. Operationally, the pattern match selects the first case (s_i, e_i) such that $a \in s_i$, and evaluates the expression e_i. An inexhaustive match is a type error.

The "alloc" operation is used for allocation of tuples, constructors, arrays, or **data** arrays.

The array operations define primitives to load and store values in arrays. The store operation is the only non-functional primitive in the language.

The "assert" statement asserts that a predicate must be true. The runtime uses these predicates to validate array bounds, and other runtime properties.

3.3 Phobos

The Phobos parser provides dynamic and extensible parsing. Languages can be augmented with new syntax and semantics, and added to the system runtime dynamically.

The central issue in an extensible parser is the representation of semantic actions—the programs that describe, for each clause in the grammar, how to form the abstract syntax tree. Our approach is to represent all intermediate forms as terms, and to use the MetaPRL term rewriter to define semantic actions.

For example, Figure 5 shows the language description of simple arithmetic expressions including factorials. The entire description is represented as a language module, which can be incrementally refined and extended in inheriting modules. Based on this language module, Phobos can lex and parse a source string and return a MetaPRL term that encodes its semantic meaning.

A Phobos language module consists of

- *Term declarations*: importing terms from existing MetaPRL modules. In the above example, the arithmetic meta operations are imported from `Base_meta`, a standard MetaPRL module that defines basic

Definition	Description
$a ::= \mathbf{nil}(t)$	The "nil" value for type t
$\mid \mathbf{boxed}(i)$	Boxed integer constants
$\mid i$	Native integer constants
$\mid x$	Floating-point constants
$\mid v$	Variables
$unop ::= - \mid\, ! \mid \cdots$	Unary operations
$binop ::= + \mid - \mid * \mid / \mid \cdots$	Binary operations
$s ::= [i_1^1, i_2^1], \ldots, [i_1^n, i_2^n]$	Integer interval set
$c ::= s \to e$	Match case
$p ::= bounds(v[a_1 \ldots a_2])$	Bounds check
$\mid is\text{-}pointer(v[a])$	Pointer check
$\mid is\text{-}function(v[a])$	Function check
$alloc ::= \langle a_1, \ldots, a_n \rangle : t$	Tuple allocation
$\mid \mathbf{const}(v[a_1, \ldots, a_n], i) : t$	Constructor allocation
$\mid array(a_1, a_2, :)t$	Array allocation
$\mid malloc(a)$	Malloc
$e ::= \mathbf{let}\ v : t = unop\ a\ \mathbf{in}\ e$	Unary operations
$\mid \mathbf{let}\ v : t = a_1\ binop\ a_2\ \mathbf{in}\ e$	binary operations
$\mid \mathbf{let}\ v : t = (\mathbf{ext}\ \texttt{"s"} : t)(a_1, \ldots, a_n)\ \mathbf{in}\ e$	Calls to the runtime
$\mid v(a_1, \ldots, a_n)$	Tail call
$\mid \mathbf{match}\ a\ \mathbf{with}\ c_1 \mid \cdots \mid c_n$	Set membership
$\mid \mathbf{let}\ v = alloc\ \mathbf{in}\ e$	Allocation
$\mid \mathbf{let}\ v : t = a_1[a_2]\ \mathbf{in}\ e$	Load
$\mid a_1[a_2] : t \leftarrow a_3; e$	Store
$\mid assert(p); e$	Assertion
$\sigma ::= v : t$	Variable type
$\mid f : t = (v_1, \ldots, v_n) \to e$	Function definition
$\Sigma ::= \sigma_1, \ldots, \sigma_n$	Variable environment

Fig. 4. FIR expressions

operations on numbers and simple conversions for their simplification. Term declarations serve the purpose of verification and proper scoping within MetaPRL modules. Terms do not have to be declared if they are explicitly named with their parent module, for example `Itt_int_base!number`.

- *Lexical information*: terminal symbols are named and defined by their corresponding regular expressions. Upon successfully matching a regular expression, the resulting token is represented as `__token__[p:s]{'pos}`, where `p` stores the matched string, and `'pos` its source position. This term can be given further meaning by an optional lexical rewrite. In the example, numbers are rewritten to MetaPRL number terms.
- *Precedence rules*: used to define precedence and associativity of terminal symbols and production rules.
- *Grammar*: Expressed in BNF, each production may contain a list of rewrites that define the corresponding semantic action. If more than one rewrite is given, the first matching rewrite is carried out during parsing. If no rewrite is given, a default rewrite is used that builds a tuple term from the right-hand side.
- *Post-parsing rewrites*: Possibly multiple sections of rewrites that are executed in sequential order after parsing. In the above example, the two rewrites are responsible for replacing a `fact` term with its actual value by unfolding factorials into multiplications. At the time of applying these rewrites, the MetaPRL refiner contains several "built-in" conversions that, for example, reduce the meta arithmetic terms.
- *Optimizations*: Optional target patterns for optimizations.

```
Module Calculator

Terms -extend "Base_meta" {
   declare meta_sum{'e1; 'e2}, meta_diff{'e1; 'e2}
   declare meta_prod{'e1; 'e2}, meta_quot{'e1; 'e2}
}

Terms -extend "@" {
   declare fact{'e}
}

Tokens -longest {
   NUM = "[1-9][0-9]*"        { __token__[p:s]{'pos} -> Itt_int_base!number[p:n] }

   TIMES = "*"                {}
   DIV = "/"                  {}
   PLUS = "+"                 {}
   MINUS = "-"                {}

   LPAREN = "("               {}
   RPAREN = ")"               {}
   BANG = "!"                 {}

   * EOL = "\\n"              {}
   * SPACE = " "              {}
}

%left PLUS MINUS
%left TIMES DIV
%left LPAREN RPAREN
%left BANG

Grammar -start e {
   e ::= NUM                  {}
      | e PLUS e              { 'e1 PLUS 'e2     -> meta_sum{'e1; 'e2} }
      | e MINUS e             { 'e1 MINUS 'e2    -> meta_diff{'e1; 'e2} }
      | e TIMES e             { 'e1 TIMES 'e2    -> meta_prod{'e1; 'e2} }
      | e DIV e               { 'e1 DIV 'e2      -> meta_quot{'e1; 'e2} }
      | e BANG                { 'e BANG          -> fact{'e} }
      | LPAREN e RPAREN       { LPAREN 'e RPAREN -> 'e }
}

Rewrites {
   fact{1} -> 1
   fact{'number} -> meta_prod{'number; fact{meta_diff{'number; 1}}}
}
```

Fig. 5. A grammar for simple arithmetic with factorials.

Given our language module, `1+2+3+4!` yields `Itt_int_base!number[30:n]`, a MetaPRL number term representing the number 30.

4 System Architecture

There are several technical issues in integrating these systems. The first issue is defining a shared language for MetaPRL and Mojave (Phobos and MetaPRL already share a common language of terms). Next, in order for MetaPRL to reason about Mojave programs, we have to formalize the language, including its operational semantics.

4.1 FIR as a common language

We are using the FIR as the common MetaPRL/Mojave language, for several reasons. First, all the front-ends, including C, ML, and Java produce programs in FIR; if we can reason about the FIR, we can reason about programs produced by any of these languages. Second, the FIR has a precise semantics, where many of the source languages do not (for example, C).

However, the disadvantage of using the FIR is that it is difficult in general to translate source-level specifications and proofs to their corresponding specifications and proofs at the FIR level. The optimization problem is not nearly so hard, and much of our current work has been developing operational reasoning in MetaPRL about programs in the FIR.

The Mojave compiler does not use terms for its internal representation of programs. For communication with MetaPRL we develop "glue" code to translate between the Mojave FIR representation of a program, and the MetaPRL term representation of the program. This glue code is straightforward; for the remainder of the paper, we will assume programs are represented as terms.

4.2 FIR term representation

The MetaPRL term representation for FIR programs is straightforward. In most cases, the term that represents an FIR expression has an explicit operator name (opname), and a set of subterms described recursively. We illustrate the translation with a few examples, using the notation $[\![\cdot]\!]$ for the term representation of a FIR program.

The atom values tagged with a name and any additional parameters.

$$\begin{aligned} [\![v]\!] &= \mathsf{atomVar}\{v\} \\ [\![i]\!] &= \mathsf{atomInt}\{i\} \\ [\![i_{32,signed}]\!] &= \mathsf{atomRawInt}\{\mathsf{int32};\ \mathsf{signedInt};\ i\} \end{aligned}$$

Expressions are a bit more interesting because of their binding structure. The term representation of an expression, in contrast to the ML representation, uses explicit binding in the form of higher-order abstract syntax [15]. As Pfenning mentions, the advantage of higher-order abstract syntax is that substitution and alpha-renaming are automatic. The disadvantage is that analyses that modify the binding in unusual ways become difficult to define. We illustrate the term syntax with the term for binary arithmetic.

$$[\![\textbf{let}\ v : t = a_1\ binop\ a_2\ \textbf{in}\ e]\!] = \mathsf{letBinop}\{[\![t]\!]; [\![binop]\!]; [\![a_1]\!]; [\![a_2]\!]; v.[\![e]\!]\}$$

The remaining terms follow the same general form, and we omit them here.

4.3 Operational semantics

The operational semantics of the FIR is defined using rewriting rules in MetaPRL. The actual operational definition is quite large because there are many combinations of arithmetic operations and values. However, the forms of definition are straightforward. For example, the operational rule for addition has the following general form, which we write using simplified pretty-printed notation. To be faithful to the implementation, we are using modular arithmetic.

$$(\mathbf{let}\ v\colon t = \mathsf{atomInt}\{i\} + \mathsf{atomInt}\{j\}\ \mathbf{in}\ e[v]) \rightarrow e[i+j]$$

The control operator match has a more interesting definition. The match operator is a pattern match of a number i against a set of intervals. The number of interval cases is arbitrary, and reduction performs one case analysis at a time.

$$\begin{gathered}(\mathbf{match}\ i\ \mathbf{with}\ s \rightarrow e \mid cases) \\ \longrightarrow (\mathbf{if}\ i \in s\ \mathbf{then}\ e\ \mathbf{else}\ \mathbf{match}\ i\ \mathbf{with}\ cases)\end{gathered}$$

The interval s is represented as a list of closed intervals $[i_{11}, i_{12}], \ldots, [i_{n1}, i_{n2}]$, and the membership operation is defined inductively.

$$\begin{gathered}(i \in ([j,k] :: interval)) \rightarrow (i \leq j \wedge i \leq k) \vee (i \in interval) \\ (i \in []) \rightarrow false\end{gathered}$$

Once again, the remainder of the operational semantics is straightforward, and we do not present it here.

4.4 Models and usage

The question of *models* is probably the most interesting topic in this translation. Ideally, we would develop a model of the FIR in a type theory or other higher-order logic, and then *prove* the operational semantics and typing rules. Note that a complete model would need to represent both partial functions and general recursive types. We have not developed this model, and we presume that it is likely that we will need to restrict validation to a fragment of the calculus that has a well-defined formal model. In the meantime, we treat the operational semantics axiomatically.

As Figure 1 illustrates, there are currently two major ways that we use the MetaPRL/Mojave system. The (i) path uses the Mojave front-ends to generate FIR code, which is then passed to MetaPRL for optimization. The (ii) path produces FIR from a Phobos description, optimizes it, and passes it to the compiler for code generation.

5 Examples

We illustrate the system with two optimizations. The MetaPRL/Mojave systems, including examples, can be downloaded from `http://www.metaprl.org` and `http://mojave.cs.caltech.edu`.

5.1 Dead-code elimination

Some standard code transformations are incredibly easy to define using term rewriting. Dead-code elimination is one of the simplest. The idea of dead-code elimination is to remove any code that does not affect the result of a computation. The problem is not computable in general, although we can develop proof procedures to catch a fairly broad set. The usual approximation is to use a *syntactic* characterization: the sub-expression e_1 in $\mathbf{let}\ v\colon t = e_1\ \mathbf{in}\ e_2$ is *dead* if v is not free in e_2. Second-order term rewriting makes this easy to characterize. The following rewrites can be derived as theorems in MetaPRL:

$$\begin{aligned}(\mathbf{let}\ v\colon t = unop\ a\ \mathbf{in}\ e) &\rightarrow e \\ (\mathbf{let}\ v\colon t = a_1\ binop\ a_2\ \mathbf{in}\ e) &\rightarrow e \\ (\mathbf{let}\ v = alloc\ \mathbf{in}\ e) &\rightarrow e \\ (\mathbf{let}\ v\colon t = a_1[a_2]\ \mathbf{in}\ e) &\rightarrow e\end{aligned}$$

Dead-code elimination is then performed by normalizing the program with respect to these rewrites. Note that the expression e in the redex does not mention the variable v, which means that v is not allowed to appear free in e (the second-order pattern $e[v]$ would have allowed v to appear in e, and would not be provable [13]). Also, note that the first-order definition using substitution would not be as useful for dead-code elimination because the rule does not specify explicitly that the variable v is dead.

$$(\textbf{let}\ v : t = \mathit{unop}\ a\ \textbf{in}\ e) \rightarrow e[a/v]$$

There are two main differences between this formal dead code elimination (using path *(ii)* in Figure 1), and the standard dead-code elimination (using path *(i)*). First, the formal definition is much smaller – the Mojave dead-code elimination phase is some 700 lines of OCaml code. Second, the OCaml implementation is also much more general, because it makes use of global program properties. For example, the OCaml implementation performs dead-argument elimination, where a function parameter can be eliminated if it is never used. This requires modification of all calls to the function through the program, a global operation that is difficult to perform using term rewriting.

5.2 Partial evaluation

The next example illustrates a simple, but non-trivial, application of partial evaluation. Consider the following FIR code (we omit the types for clarity). The *power* function computes the value $res * x^y$, and passes it to the continuation *cont*. The *power5* function computes the specific case where $res = 1$ and $y = 5$.

```
let power (res, x, y, cont) =
    if y = 0 then
        cont(res)
    else
        let res' = res * x in
        let y' = y - 1 in
            power(res', x, y', cont)

let power5 (x, cont) =
    power(1, x, 5, cont)

inline power(res, x, number[i], cont)
```

For this example, we would like to "unroll" the definition of *power5* to a sequence of 4 multiplications $x * x * x * x * x$. The programming language, defined in Phobos, includes the **inline** extension where the programmer can indicate *patterns* that should be expanded using the **inline** keyword. For the example above, the inline instruction specifies that a call to the *power* function should be expanded when its third argument is a number.

Based on this information, the MetaPRL system constructs a rewrite to force the unfolding.

```
let power (res, x, number[i], cont) ⟶
    if number[i] = 0 then
        cont(res)
    else
        let res' = res * x in
        let y' = number[i] - 1 in
            power(res', x, y', cont)
```

Next, *power5* is normalized relative to the rewrite, and all calls to the *power* function with a constant exponent are inlined. The final definition of the *power5* function is as follows.

```
let power5(x, cont) =
    let x1 = x * x in
    let x2 = x1 * x in
    let x3 = x2 * x in
    let x4 = x3 * x in
        cont(x4)
```

The optimized code produced by MetaPRL is still suboptimal; if we assume that multiplication is associative, the number of multiplications can be reduced to three. We have not implemented partial evaluation as a compiler phase in path *(i)*. Partial evaluation is most naturally expressed using the operational semantics of the program; any implementation would needlessly reimplement program evaluation.

6 Conclusion and Future Work

The work presented in this paper demonstrates the *principle* of formal integrated design environments, but the integration is far from complete. Among the next steps are: 1) a complete formalization of the operational semantics and type system for the FIR, 2) a more general framework for performing partial evaluation. The Mojave compiler architecture has many (around 30) stages of program transformation and optimization. It seems likely that many of these transformations can be significantly simplified by implementing them in the theorem prover. Another important direction is "bootstrapping." Currently, MetaPRL is still layered over the OCaml compiler because the Mojave implementation of ML does not include a module system. Removal of this obstacle would enable complete integration of the formal design environment.

References

1. Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 215 224. IEEE, June 1987.
2. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proc. of Fifth Symp. on Logic in Comp. Sci.*, pages 95 197. IEEE, June 1990.
3. Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, 1998. Second Edition.
4. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, April 1995. `http://www.csl.sri.com/sri-csl-fm.html`.
5. M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.
6. Jason Hickey. Fault-tolerant distributed theorem proving. In Harald Ganzinger, editor, *Automated deduction CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 227 231, Trento, Italy, July 7-10, 1999. Springer-Verlag.
7. Jason J. Hickey. *The MetaPRL Logical Programming Environment.* PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, 2001.
8. Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`, 2002.
9. S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994.
10. S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445-484, 1997.
11. Christoph Kreitz, Mark Hayden, and Jason Hickey. A proof environment for the development of group communication systems. In C. Kirchner and H. Kirchner, editors, *15^{th} International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 317-332. Springer Verlag, 1998.
12. Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17^{th} ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 34 of *Operating Systems Review*, pages 80 92, 1999.
13. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In *Theorem Proving in Higher-Order Logics (TPHOLs '02)*, 2002.
14. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer LNCS 828, 1994.
15. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199 208, Atlanta, Georgia, June 1988.
16. Didier Rémy and Jérôme Vouillon. Objective ML: A simple object oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40 53, 1997.

Reflecting Higher-Order Abstract Syntax in Nuprl

Eli Barzilay and Stuart Allen

Cornell University
{eli,sfa}@cs.cornell.edu

Abstract. This document describes part of an effort to achieve in Nuprl a practical reflection of its expression syntax. This reflection is done at the granularity of the operators; in particular, each operator of the syntax is *denoted* by another operator of the same syntax. Further, the syntax has binding operators, and we organize reflection not around the concrete binding syntax, but instead, around the abstract higher-order syntax. We formulate and prove the correctness of a core rule for inferring well-formedness of instances of operator-denoting operators.

1 Introduction

This work is part of an overall effort to get a practical reflection of syntax, computation and proof in Nuprl [4, 1, 3]. Reflecting syntax in a logical system entails writing proof rules that express that reflection, i.e. establishing an inferential connection between the actual syntax used and the meta-terms supposedly referring to it.

Operator-denoting operators are called *shifted* operators: if an operator x denotes operator y, then x is called a shifted y, and will be typeset as $\underline{y}$. For example, $a \underline{+} b$ denotes $c + d$ if a denotes c and b denotes d. The plus operator denotes a function that takes two integers and returns an integer, and its shifted version denotes a function that takes two terms and returns a term. The problem is what do we do with an operator that has a bound subterm: for example, $\forall x.\ P(x)$ is an operator that denotes a function taking a boolean or propositional function and returning a boolean or a proposition (its syntactic form is, of course, binding).

The obvious choice for the semantics of the shifted version would be a function, $\underline{\forall}(x, P)$ that takes two expressions as input values: one for the bound name, and one for the body, and constructs the concrete $\forall$ term. We will not pursue this direction. Instead, we shall adopt a higher-order abstract syntax [7]. Going in this direction, we get the usual benefits of this approach over concrete syntax (or alternatives like de-Bruijn indexes), such as specified in [8]. But we get a further bonus: it allows us to retain the same binding structure as the operator being denoted. In particular, the single input argument for $\underline{\forall}$ has the same binding as $\forall$: it takes in a term-valued function as an argument.

Implementing reflection in a programming language is usually done in a straightforward way: simply expose the implementation's evaluation function so it is available to programs written in the language. However, in a logical setting this is usually not the chosen approach, and the result is usually limited in its usability to theoretical or toy examples. The best example is Gödel numbers [5] which are good as a theoretical tool but not fit for an actual running system. Our goal is an eventual implementation that follows the same principle of exposing internal functionality: this is the outcome of operators being denoted by operators. The result is expected to be a system that has practical reflection implemented as is the situation in programming languages.

This construction is intended for the Nuprl system, but we avoid relying on a specific substitution function, which makes this approach applicable in the general case. Relevant information about Nuprl terms is limited to their content: a Nuprl term contains an operator id, and a list of bound subterms, each containing a list of bound variables and a term. Throughout this text we use a more conventional notation, with the extension of using underlines for shifted operators.

Returning to the question above: we begin by asking what is the semantics of $\underline{\forall}$? The semantics of a *concrete* shifted $\forall$ is the trivial one given above, but the semantics for $\underline{\forall}$ is more subtle.

2 Semantics of Shifted Operators

Since $\underline{\forall}$ is a binding operator, it takes a function as an argument. Our basic requirement is that $F(t)$ be the result of the 'All-Instantiation' rule applied to $\underline{\forall} x.\ F(x)$ and t. This means that F needs to be a *substitution* *function*. So the semantics we adopt for $\underline{\forall}x.\ F(x)$ is that it denotes the $\forall$ formula whose predicate part is $F(u)$ and whose binder is u for some u — almost.

But which u? As usual we can avoid this question by using a higher-order abstract syntax, and say that what is denoted is actually the α-equivalence class of all such formulas where some appropriate u could be found. From this point forth, we use 'Term' to refer to these α-equivalence classes rather than the concrete terms.

Before going to the technical parts, lets consider how we might reason about this in the reflective logic. The first intuition is that proving that something is a Term depends only on having a quoted operator $\underline{\mathsf{opid}}$ and on its subparts in a simple compositional way:

$$\begin{array}{l} \vdash \underline{\mathsf{opid}(\overline{v_1}.b_1;\overline{v_2}.b_2;\ldots;\overline{v_n}.b_n)} \in \mathsf{Term} \\ \text{if } \overline{v_1} : \mathsf{Term} \vdash b_1 \in \mathsf{Term} \\ \quad\ \overline{v_2} : \mathsf{Term} \vdash b_2 \in \mathsf{Term} \\ \quad\ \vdots \\ \quad\ \overline{v_n} : \mathsf{Term} \vdash b_n \in \mathsf{Term} \end{array}$$

This seems fine, but it fails with bound variables. For example, the following can be proved:

$$\begin{array}{l} \vdash \underline{\lambda(x.\ \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2})} \in \mathsf{Term} \\ \text{because } x : \mathsf{Term} \vdash \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2} \in \mathsf{Term} \end{array}$$

The premise line is trivial, but the original statement is false, because the quoted $\underline{\lambda}$-term contains a function which is not a substitution function — it is not a "template" function. In other words, there is no literally quoted term that this value stands for.

When inspecting this term, we can compare it to similar but valid terms to see what went wrong with this rule:

1. $\underline{\lambda(x.\ \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ 2)}$
2. $\underline{\lambda(x.\ \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{1})}$
3. $\lambda x.\ \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2}$

The two $\underline{\lambda}$-terms are fine, because they're built from substitution functions, and the last one is a simple Term $\rightarrow$ Term function. The difference between these terms and the previous one indicates what is wrong with the above rule: the bound variable should not be used as a value. It is a binding that should only be used in template holes, as there is no real value that this variable is ever bound to that can be used. In the valid examples, the first one did not *use* the bound value except for sticking it in its place. The second one almost used the value, but since the two branches are identical it is possible to avoid evaluating the test term; therefore it can be evaluated without using it, and the last one is not a Term but a function on Terms, so it can use that value as usual.

The conclusion is that a bound variable can be used only as an argument of a quoted term constructor. In other words, it can serve only as a value that is "computationally inert", much like universe expressions in [2]. This is also similar to variables that are bound by Scheme's syntax rules [6] — they are template variables that can be used in syntactic structures to build new structures[1]. When put in this light, it seems that any attempt to get this property in a proof fails. The lesson from this is: variables bound by quoted operators do not behave like normal bindings in the sense that they do not provide any values usable on the normal Nuprl level — and this is also true regarding universe expressions.

3 Term Definition

We take CTerms as concrete terms: the type of objects intended to be ordinary syntax objects with binding operators. A more precise definition is given later, in Section 5. To define the Term type, we also need to

[1] For example, in the template `((foo x) (bar x))`, the identifier '`x`' is just a place holder that can be used to stick a value in a template; it is not possible to inspect its value.

introduce a predicate, `is_subst`, which is used to distinguish proper substitution functions. This predicate is defined in Section 6, and it has specific rules which are introduced in Section 6.1.

As said above, Terms are defined over these CTerms:

$$\mathsf{Term} \equiv \mathsf{CTerm}//\alpha$$

Terms are constructed by shifted operators, which have the semantics of functions that create Terms from Term substitution functions. For example:

$$\underline{\lambda} : \{f : \mathsf{Term} \to \mathsf{Term} \mid \mathtt{is\_subst}_1(f)\} \to \mathsf{Term}$$

using a version of `is_subst` that works with one argument functions. Generally, $\mathtt{is\_subst}_n$ is a predicate over $\mathsf{Term}^n \to \mathsf{Term}$. To simplify things, we drop the n when the context is clear.

CVar is a subsets of CTerm, which contains only atomic variable terms. Correspondingly, $\mathsf{Var} = \{\{x\} \mid x \in \mathsf{CVar}\}$, therefore $\mathsf{Var} \subseteq \mathsf{Term}$, since variables are α-equivalent only to themselves. Two assumptions that will be used in the following are that we have an infinite supply of distinct variables in CVar (and therefore in Var) and that there is at least one closed CTerm we can use.

4 Operations, Assumptions, and Facts

These are the operations that will be needed in the following text:

$\cdot\lceil$ taking the α-equivalence class of an object.
$\cdot\lfloor$ choosing an element of an α-equivalence class. This is some function, such as one that chooses the first available variable names using lexicographic order.
$\cdot[\cdot/\cdot]$ standard capture-avoiding substitution on CTerms. It can be used to substitute for multiple variables at one shot, provided that the number of supplied terms matches the number of variables, which are all distinct.
$\cdot[\![\cdot/\cdot]\!]$ substitution for Terms, which is defined using the above operations as: $b[\![\overline{x}/\overline{v}]\!] = b\lfloor[\overline{x}\lfloor/\overline{v}\lfloor]\lceil$.
$\mathrm{newcvar}(\cdot)$ returns a new CVar, i.e., $\mathrm{newcvar}(t)$ is neither free nor bound in $t \in \mathsf{CTerm}$.
$\mathrm{newvar}(\cdot)$ is similar to $\mathrm{newcvar}(\cdot)$ but for Terms, defined as: $\mathrm{newvar}(x) = \mathrm{newcvar}(x\lfloor)\lceil$.
$\mathrm{newcvar}_n(\cdot)$ returns n new CVars, defined as:
$\mathrm{newcvar}_1(x) = \mathrm{newcvar}(x)$,
$\mathrm{newcvar}_{n+1}(x) = (\mathrm{let}\ v == \mathrm{newcvar}_n(x)\ \mathrm{in}\ v, \mathrm{newcvar}(v, x))$.
$\mathrm{newvar}_n(\cdot)$ returns n new Vars, defined in the same way as $\mathrm{newcvar}_n(\cdot)$.

We use versions of these operations that are generalized to any lists and tuples of input arguments in an obvious way. The $\mathrm{newcvar}(\cdot)$ and $\mathrm{newvar}(\cdot)$ operations are further extended to functions by plugging in some closed dummy term argument (that we name '0') and using the result:

$$\forall f : \mathsf{Term}^n \to \mathsf{Term}.\ \mathrm{newvar}_m(f) = \mathrm{newvar}_m(f(0^n))$$

Below we will often justify things of the form $a\lceil = b\lceil$, by mentioning lemmas of the form $a =_\alpha b$, without emphasizing this transition.

Note: an overline indicates the value is a tuple and a way to index its elements. For example, $\overline{x} : \mathsf{Var}^n$ means that $\overline{x}$ is a list of n Vars, and that x_i is the ith element of this list; that is, x is a function from $i : 1 \ldots \mathrm{len}(\overline{x})$ to the ith element of $\overline{x}$.

4.1 Important Assumptions and Facts

In this section we state several assumptions and derived facts about substitution — the assumptions are not argued for, but we think that it is clear they are all true for any reasonable definition of substitution (one that respects the usual term binding structure). This allows us to take substitution as given and avoid getting into a specific implementation. These will be used in the following text.

$\star 1$ $\forall x : \mathsf{Term}.\ x = x\downharpoonleft\upharpoonright$
$\star 2$ $\forall x : \mathsf{CTerm}.\ x =_\alpha x\upharpoonright\downharpoonleft$
This fact is mostly used when nested in a bigger term, see $\star 4$ below.
$\star 3$ $\forall x : \mathsf{CVar}.\ x = x\upharpoonright\downharpoonleft$
because $x \in \mathsf{CVar} \ \Rightarrow\ x\upharpoonright\downharpoonleft = \{x\}\downharpoonleft = x$.
$\star 4$ $\forall \overline{x_1}, \overline{x_2} : \mathsf{CTerm}^n, \overline{v} : \mathsf{CVar}^n,\ b : \mathsf{CTerm}.\ \overline{x_1} =_\alpha \overline{x_2} \ \Rightarrow\ b[\overline{x_1}/\overline{v}] =_\alpha b[\overline{x_2}/\overline{v}]$
Note that using this fact, $\star 2$ can be used in a subterm of an α-equality, since: $\forall t, x : \mathsf{CTerm}.\ t =_\alpha t[x\upharpoonright\downharpoonleft/x]$
$\star 5$ $\forall b_1, b_2 : \mathsf{CTerm},\ \overline{t} : \mathsf{CTerm}^n,\ \overline{v} : \mathsf{CVar}^n.\ b_1 =_\alpha b_2 \ \Rightarrow\ b_1[\overline{t}/\overline{v}] =_\alpha b_2[\overline{t}/\overline{v}]$
note that $\overline{v}$ is the same on both sides (free variables in the body are not changed).
$\star 6$ $\forall t : \mathsf{CTerm},\ \overline{x_1} : \mathsf{CTerm}^{n_1},\ \overline{x_2} : \mathsf{CTerm}^{n_2},\ \overline{v_1}, \overline{u} : \mathsf{CVar}^{n_1},\ \overline{v_2} : \mathsf{CVar}^{n_2}$.
 the sequence $\overline{v_1}, \overline{v_2}$ are distinct & $\overline{u}$ are distinct, not free in $t, \overline{x_2}$
 $\Rightarrow\ t[\overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}] =_\alpha t[\overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}][\overline{x_1}/\overline{u}]$
This is simple to verify:

$$\begin{aligned} & t[\overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}][\overline{x_1}/\overline{u}] \\ \text{(any of } \overline{u} \text{ do not occur free in } t) &=_\alpha t[\overline{u}[\overline{x_1}/\overline{u}], \overline{x_2}[\overline{x_1}/\overline{u}]/\overline{v_1}, \overline{v_2}] \\ (\overline{u} \text{ are distinct}) &=_\alpha t[\overline{x_1}, \overline{x_2}[\overline{x_1}/\overline{u}]/\overline{v_1}, \overline{v_2}] \\ \text{(any of } \overline{u} \text{ do not appear in } \overline{x_2}) &=_\alpha t[\overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}] \end{aligned}$$

Note that it is easy to show that such a $\overline{u}$ exists by choosing it as:
 let $\overline{u} = \mathrm{newcvar}_{n_1}(t, \overline{x_2}, \ldots)$
$\star 7$ $\forall t : \mathsf{Term},\ \overline{x_1} : \mathsf{Term}^{n_1},\ \overline{x_2} : \mathsf{Term}^{n_2},\ \overline{v_1}, \overline{u} : \mathsf{Var}^{n_1},\ \overline{v_2} : \mathsf{Var}^{n_2}$.
 the sequence $\overline{v_1}, \overline{v_2}$ are distinct & $\overline{u}$ are distinct, not free in $t, \overline{x_2}$
 $\Rightarrow\ t\llbracket \overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}\rrbracket = t\llbracket \overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}\rrbracket\llbracket \overline{x_1}/\overline{u}\rrbracket$
Again, verifying this is simple: from $\star 6$ we know that
 $t\downharpoonleft[\overline{u}\downharpoonleft, \overline{x_2}\downharpoonleft/\overline{v_1}\downharpoonleft, \overline{v_2}\downharpoonleft][\overline{x_1}\downharpoonleft/\overline{u}\downharpoonleft]\upharpoonright = t\downharpoonleft[\overline{x_1}\downharpoonleft, \overline{x_2}\downharpoonleft/\overline{v_1}\downharpoonleft, \overline{v_2}\downharpoonleft]\upharpoonright$,
so:

$$\begin{aligned} t\llbracket \overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}\rrbracket\llbracket \overline{x_1}/\overline{u}\rrbracket &= t\downharpoonleft[\overline{u}\downharpoonleft, \overline{x_2}\downharpoonleft/\overline{v_1}\downharpoonleft, \overline{v_2}\downharpoonleft]\upharpoonright\downharpoonleft[\overline{x_1}\downharpoonleft/\overline{u}\downharpoonleft]\upharpoonright \\ (\star 2,\ \star 5) &= t\downharpoonleft[\overline{u}\downharpoonleft, \overline{x_2}\downharpoonleft/\overline{v_1}\downharpoonleft, \overline{v_2}\downharpoonleft][\overline{x_1}\downharpoonleft/\overline{u}\downharpoonleft]\upharpoonright \\ (\text{by the use of } \star 6 \text{ above}) &= t\downharpoonleft[\overline{x_1}\downharpoonleft, \overline{x_2}\downharpoonleft/\overline{v_1}\downharpoonleft, \overline{v_2}\downharpoonleft]\upharpoonright \\ &= t\llbracket \overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}\rrbracket \end{aligned}$$

A similar note holds here: it is easy to show that such a $\overline{u}$ exists if it is chosen as:
 let $\overline{u} = \mathrm{newcvar}_{n_1}(t\downharpoonleft, \overline{x_2}\downharpoonleft, \ldots\downharpoonleft)\upharpoonright = \mathrm{newvar}_{n_1}(t, \overline{x_2}, \ldots)$
$\star 8$ $\forall c : \mathsf{CTerm},\ \overline{v}, \overline{u} : \mathsf{CVar}^n,\ \overline{s}, \overline{t} : \mathsf{CTerm}^n$.
 $\overline{u}$ are not free in c except for $\overline{v}$ $\Rightarrow\ c[\overline{s}/\overline{v}][\overline{t}/\overline{u}] =_\alpha c[\overline{s}[\overline{t}/\overline{u}]/\overline{v}]$
Note that the $\overline{v}$ exception is usually not needed.
$\star 9$ $\forall c : \mathsf{Term},\ \overline{v}, \overline{u} : \mathsf{Var}^n,\ \overline{s}, \overline{t} : \mathsf{Term}^n$.
 $\overline{u}$ are not free in c except for $\overline{v}$ $\Rightarrow\ c\llbracket \overline{s}/\overline{v}\rrbracket\llbracket \overline{t}/\overline{u}\rrbracket = c\llbracket \overline{s}\llbracket \overline{t}/\overline{u}\rrbracket/\overline{v}\rrbracket$
This is easily shown by $\star 2$ and the definition of $\cdot\llbracket\cdot/\cdot\rrbracket$, using the previous fact.

A general intuition that arises from these facts and others, is that Term values are indeed isomorphic to CTerms: as long as there are no "dirty" concrete tricks played by *using* names of bound variables, facts that hold for CTerms will have corresponding versions for Terms.

5 Definitions of Shifted Operators

In the general case, a *shifted* operator id, opid, is defined as a function that takes in some substitution functions (determined by `is_subst`) of some arities, and returns a Term value. This is done in the obvious way: each of the substitution functions is used to plug in new variables; then the results, with the chosen variables, are all packaged into a CTerm; and finally, the α-equivalence class of this result produces the resulting Term. The actual representation is not too important — we could go with pairs and lists. For example:

$$\underline{\lambda}(f) = \langle \text{`}\lambda\text{'}, [\langle [\mathrm{newvar}(f)\downharpoonleft], f(\mathrm{newvar}(f))\downharpoonleft\rangle]\rangle\upharpoonright$$

but this gets too complex in the general case (and it makes analysis hard, since we should know if a pair stands for a bound term, a term, or a pair of terms).

Instead, we use some types and abstract operations, which avoids committing us to some representation. The additional types we need are:

- OpId will be used for term name labels;
- $\mathsf{BndCTerm}_a$ is a *bound CTerm* (where $a : \mathbb{N}$) -- packaging a CTerm with a distinct CVars.

$\mathsf{BndCTerms}$ are created with a `mkBndCTerm` constructor[2]:

$$\texttt{mkBndCTerm} \in a : \mathbb{N} \to (1 \ldots a \to \mathsf{CVar}) \to \mathsf{CTerm} \to \mathsf{BndCTerm}_a$$

An alternate syntax for `mkBndCTerm` can be more natural when a is known:

$$\texttt{mkBndCTerm}(\overline{x}, t) \quad \text{stands for} \quad \texttt{mkBndCTerm}(\text{len}(\overline{x}), (\lambda i.\, x_i), t)$$

CTerms are created with `mkCTerm`:

$$\begin{aligned}\texttt{mkCTerm} \in\ & \mathsf{OpId} \to n : \mathbb{N} \\ & \to a : (1 \ldots n \to \mathbb{N}) \\ & \to (i : 1 \ldots n \to \mathsf{BndCTerm}_{a_i}) \\ & \to \mathsf{CTerm}\end{aligned}$$

An alternate syntax for this which can be more natural when n, a are known is:

$$\texttt{mkCTerm}(o,\ [\texttt{mkBndCTerm}(\overline{x_1}, t_1), \ldots, \texttt{mkBndCTerm}(\overline{x_n}, t_n)])$$

which can be used instead of

$$\texttt{mkCTerm}(o,\ n,\ (\lambda i.\, \text{len}(\overline{x_i})),\ (\lambda i.\, \texttt{mkBndCTerm}(\overline{x_i}, t_i)))$$

The next thing we need is a type which is the subset of $\mathsf{Term}^n \to \mathsf{Term}$ functions that are substitution functions (using the `is_subst` predicate):

$$\mathsf{SubstFunc}_n = \{f : \mathsf{Term}^n \to \mathsf{Term} \mid \texttt{is\_subst}_n(f)\}$$

Now we have reached the point where we can finally define a `mkTerm` constructor for Terms which uses `mkCTerm`:

$$\begin{aligned}\texttt{mkTerm} \in\ & \mathsf{OpId} \to n : \mathbb{N} \\ & \to a : (1 \ldots n \to \mathbb{N}) \\ & \to (i : 1 \ldots n \to \mathsf{SubstFunc}_{a_i}) \\ & \to \mathsf{Term}\end{aligned}$$

This function is defined as:

$\texttt{mkTerm}(o, n, a, f) =$
$\quad \texttt{mkCTerm}(o, n, a, \lambda i.\ \text{let } \overline{x} = \text{newvar}_{a_i}(f_i) \text{ in } \texttt{mkBndCTerm}(\overline{x}\lfloor, f_i(\overline{x})\rfloor))\rceil$

and the alternate syntax for this is:

$$\texttt{mkTerm}(o, [\langle a_1, f_1\rangle, \ldots, \langle a_n, f_n\rangle])$$

which stands for

$$\texttt{mkTerm}(o, n, \lambda i.\, a_i, \lambda i.\, f_i) = \texttt{mkTerm}(o, n, a, f)$$

[2] We use the notation $x : A \to B_x$ to denote functions on A such that $\forall x : A.\ f(x) \in B_x$, a type which is more conventionally denoted by $\Pi x : A.\ B_x$.

A shifted operator is a Term constructor which uses mkTerm with some fixed operator name and arity list. For example, '$\underline{\lambda}$' and '$\underline{\Sigma}$' are defined as:

$$\underline{\lambda}(f) = \texttt{mkTerm}(`\lambda', [\langle 1, f\rangle]), \quad \underline{\Sigma}(f, g) = \texttt{mkTerm}(`\Sigma', [\langle 0, f\rangle, \langle 1, g\rangle])$$

Note that since mkTerm is curried, a shifted operator is made by specifying the first three inputs: $\texttt{mkTerm}(o, n, a)$.

In addition to the assumptions and facts introduced in Section 4.1, we further assume the following:

$\star$10 We specify one way that substitution interacts with CTerms for all i, k, if it is true that
if v_k is free in t_i then none of $\overline{x_i}$ are free in either r_k or v_k
then[3],

$$\texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, t_i))[\overline{r}/\overline{v}] =_\alpha \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, t_i[\overline{r}/\overline{v}]))$$

To see why it is true using any reasonable definition of substitution, it is simpler to first see that a precondition that could be used is that none of $\overline{x_i}$ occur free in $\overline{r}, \overline{v}$; this is too restrictive for our future needs but the explanation is somewhat similar.
First of all, if v_k is not free in t_i, then there is no need for any restriction, since it does not have any effect on the result. Now, if it does appear in t_i, then it is enough to have two guarantees for the above to remain an α-equality: (a) if none of $\overline{x_i}$ are free in r_k then capture by $\overline{x_i}$ is impossible; (b) if v_k is not in $\overline{x_i}$, then none of the v_k will not get "screened out" in the body.

- A fact similar to this assumption also holds for Terms if none of $\overline{x_i}$ occur free in $\overline{r}, \overline{v}, \overline{f_i(\underline{0}^{a_i})}$ then:

$$\texttt{mkTerm}(o, n, a, \lambda i.\ f_i)[\![\overline{r}/\overline{v}]\!] = \texttt{mkTerm}(o, n, a, \lambda i.\ \lambda \overline{z}.\ f_i(\overline{z})[\![\overline{r}/\overline{v}]\!])$$

However, it turns out that this fact is not needed, so no proof is given.

$\star$11 A simple fact about renaming bound variables:
$\forall \overline{x_i}, \overline{z_i} : \mathsf{CVar}^n.\ \overline{x_i}$ are distinct & $\overline{z_i}$ are distinct & $\overline{z_i}$ are not free in b_i
$\Rightarrow\ \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, b_i))$
$=_\alpha \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}, b_i[\overline{z_i}/\overline{x_i}]))$

6 Defining is_subst

A function is a *substitution function* iff there exists an appropriate substitution that it is equivalent to. First, we describe this using CTerms, since we know how substitutions work on them:

$$\texttt{is\_subst}_n(f) \equiv \exists b : \mathsf{CTerm}.\ \exists \overline{v} : \mathsf{CVar}^n.\ \forall \overline{t} : \mathsf{CTerm}^n.\ f(\overline{t}\lceil) = b[\overline{t}/\overline{v}]\lceil \tag{1}$$

Note that f returns a Term which is an α-equivalence class, so we have an equality rather than an α-equality. This should be equivalent to directly using a Term argument for f:

$$\texttt{is\_subst}_n(f) \equiv \exists b : \mathsf{CTerm}.\ \exists \overline{v} : \mathsf{CVar}^n.\ \forall \overline{r} : \mathsf{Term}^n.\ f(\overline{r}) = b[\overline{r}\lfloor/\overline{v}]\lceil \tag{2}$$

We should show that $\forall b : \mathsf{CTerm},\ \forall \overline{v} : \mathsf{CTerm}^n$, the two sub-expressions are equivalent.

<u>(1) ⇒ (2)</u> Instantiate $\overline{t}$ with the chosen $\overline{r}\lfloor$:

$$f(\overline{r}) \overset{\star 1}{=} f(\overline{r}\lfloor\lceil) \overset{(1)}{=} b[\overline{r}\lfloor/\overline{v}]\lceil$$

<u>(2) ⇒ (1)</u> Instantiate $\overline{r}$ with $\overline{t}\lceil$ and we get:

$$f(\overline{t}\lceil) \overset{(2)}{=} b[\overline{t}\lceil\lfloor/\overline{v}]\lceil \overset{\star 2 \star 4}{=} b[\overline{t}/\overline{v}]\lceil$$

We can now try to use a version that has all Term types and no CTerm types, using the α-terms substitution, $\cdot[\![\cdot/\cdot]\!]$:

$$\texttt{is\_subst}_n(f) \equiv \exists b_a : \mathsf{Term}.\ \exists \overline{v_a} : \mathsf{Var}^n.\ \forall \overline{t_a} : \mathsf{Term}^n.\ f(\overline{t_a}) = b_a[\![\overline{t_a}/\overline{v_a}]\!] \tag{3}$$

Now, verify that this is indeed equivalent to the other two definitions:

[3] Note that the α-equality is needed only because the substitution definition might introduce arbitrary renamings.

<u>(2) ⇒ (3)</u> Let $b_a = b\upharpoonleft$, $\overline{v_a} = \overline{v}\upharpoonleft$, pick some $\overline{t_a}$, and instantiate $\overline{r}$ with it:

$$f(\overline{t_a}) \overset{(2)}{=} b[\overline{t_a}\downharpoonleft/\overline{r}]\upharpoonleft \overset{\star}{=} b\upharpoonleft\downharpoonleft[\overline{t_a}\downharpoonleft/\overline{v}\upharpoonleft\downharpoonleft]\upharpoonleft = b\upharpoonleft[\![\overline{t_a}/\overline{v}\upharpoonleft]\!] = b_a[\![\overline{t_a}/\overline{v_a}]\!]$$

($\star$) is true because of $\star 2$ (with b). $\star 3$ (with $\overline{v}$), and $\star 5$ (with $x_1, x_2, \bar{t}, \overline{r}$).

<u>(3) ⇒ (2)</u> Let $b = b_a\downharpoonleft$, $\overline{v} = \overline{v_a}\downharpoonleft$, pick some $\overline{r}$, and instantiate $\overline{t_a}$ with it:

$$f(\overline{r}) \overset{(3)}{=} b_a[\![\overline{r}/\overline{v_a}]\!] = b_a\downharpoonleft[\overline{r}\downharpoonleft/\overline{v_a}\downharpoonleft]\upharpoonleft = b[\overline{r}\downharpoonleft/\overline{v}]\upharpoonleft$$

6.1 The is_subst Rules

Now that we have a reasonable definition of `is_subst`, we define key rules which use `is_subst` to prove that something is a proper Term. These rules turn out to be quite simple — there are only two cases:

- $H \vdash$ `is_subst`$(x_1, x_2, \ldots, x_n.\ x_i)$
- $H \vdash$ `is_subst`$(\overline{x}.\ \underline{\text{opid}}(\overline{y_1}.\ b_1; \ldots; \overline{y_n}.\ b_n))$ where <u>opid</u> is some quoted opid
  $H \vdash$ `is_subst`$(\overline{x}, \overline{y_1}.\ b_1)$
  $H \vdash$ `is_subst`$(\overline{x}, \overline{y_2}.\ b_2)$
  ⋮
  $H \vdash$ `is_subst`$(\overline{x}, \overline{y_n}.\ b_n)$

Note that this is enough for proving the validity of any Term value: for example, quoted constants succeeds immediately since their opid is quoted and they have no subterms. Proving $t \in$ Term is achieved by showing `is_subst`$(.\ t)$. (Of course, this is not a complete set of rules, since there are more cases where we have general Term expressions that are not constants.)

6.2 Justifying the is_subst Rules

The validity of the first rule amounts to this:

$$\forall n, i : \mathbb{N}^+.\ i \leq n \ \Rightarrow\ \texttt{is\_subst}_n(\pi_n^i)$$

which is easily verified. Choose distinct $\overline{v} = v_1, \ldots, v_n$ variables, and let $b = \pi_n^i(\overline{v}) = v_i$. Then, $\forall \overline{t} : \text{Term}^n.\ \pi_n^i(\overline{t}) = v_i[\![\overline{t}/\overline{v}]\!]$ is true by the definition of π_n^i, of $\cdot[\![\cdot/\cdot]\!]$, and the distinctness of $\overline{v}$.

Our main result will be formulating and proving the validity of the second rule, but this formulation requires some preparation. First, recall that the type of `mkTerm` is:

$$\text{OpId} \to n : \mathbb{N} \to a : (1 \ldots n \to \mathbb{N}) \to (i : 1 \ldots n \to \text{SubstFunc}_{a_i}) \to \text{Term}$$

Note that, as said earlier, a shifted operator is the result of applying `mkTerm` on the first three arguments, since they define the operator symbol and the list of arities it expects. For example:

$$\underline{\lambda} = \texttt{mkTerm}(\text{`}\lambda\text{'}, 1, \langle 1 \rangle) \qquad \underline{\Sigma} = \texttt{mkTerm}(\text{`}\Sigma\text{'}, 2, \langle 0, 1 \rangle)$$

So, a shifted operator has the following type, for some given o, n, and a:

$$\texttt{mkTerm}(o, n, a) : \quad (i : 1 \ldots n \to \text{SubstFunc}_{a_i}) \to \text{Term}$$

Remember that the current goal is to conclude that for some shifted operator, <u>opid</u>:

$$\texttt{is\_subst}(\overline{x}.\ \underline{\text{opid}}(\overline{v_1}.\ b_1, \ldots, \overline{v_n}.\ b_n))$$
$$\text{if}$$
$$\texttt{is\_subst}(\overline{x}, \overline{v_1}.\ b_1)\ \&\ \ldots\ \&\ \texttt{is\_subst}(\overline{x}, \overline{v_n}.\ b_n)$$

We need to compose the <u>opid</u> function with an object that will make the result a $\mathsf{Term}^k \rightarrow \mathsf{Term}$ function (consuming the $x_1, \ldots, x_k$ variables) which we then show is a substitution function. This means the function that is composed with <u>opid</u> should get a tuple of Term^k as input and return the vector of n substitution functions, built by consuming $\overline{x}$. In short, we package all the necessary information in F:

$$F : \mathsf{Term}^k \rightarrow (i : 1 \ldots n \rightarrow \mathsf{SubstFunc}_{a_i})$$

so we get the expected:

$$\texttt{mkTerm}(o, n, a) \circ F : \mathsf{Term}^k \rightarrow \mathsf{Term}$$

Now for the main result the validity of the second rule may be formulated thus:

$$\begin{array}{l} \forall\, o : \mathsf{OpId},\ n : \mathbb{N},\ a : (1 \ldots n \rightarrow \mathbb{N}),\ k : \mathbb{N}, \\ \quad F : \mathsf{Term}^k \rightarrow (i : 1 \ldots n \rightarrow \mathsf{SubstFunc}_{a_i}). \\ \qquad \forall i : 1 \ldots n.\ \texttt{is\_subst}_{k+a_i}(\lambda_{(k)}\, ts, xs.\ F(ts)(i)(xs)) \\ \qquad\quad \Rightarrow\ \texttt{is\_subst}_k(\texttt{mkTerm}(o, n, a) \circ F) \end{array}$$

where $(\lambda_{(k)} x, y.\ B(x, y))(u_1, \ldots, u_{k+n}) \equiv B((u_1, \ldots, u_k), (u_{k+1}, \ldots, u_{k+n}))$.[4]

Proof. Assume o, n, a, k, and F are given as specified. We also assume that the constructed functions are substitution functions; therefore, for every $1 \leq i \leq n$ we get $c_i : \mathsf{Term}$, $\overline{u_i} : \mathsf{Var}^k$, $\overline{v_i} : \mathsf{Var}^{a_i}$ such that:

$$\forall r_1^1, \ldots, r_k^1, r_1^2, \ldots, r_{a_i}^2 : Term.\ F(r_1^1, \ldots, r_k^1)(i)(r_1^2, \ldots, r_{a_i}^2) = c_i\llbracket \overline{r^1}, \overline{r^2} / \overline{u_i}, \overline{v_i} \rrbracket$$

- Let $\bar{t} : \mathsf{Term}^k$ be some k Terms,
- let $\overline{x_i} = \mathrm{newvar}_{a_i}(F(\bar{t})(i))$,
- and let $\overline{s} = \mathrm{newvar}_k(\overline{c}, \overline{x_1}, \ldots, \overline{x_n})$.

Now we can proceed: our goal due to the definition of `is_subst`, is to derive an equality of the form

$$(\texttt{mkTerm}(o, n, a) \circ F)(\bar{t}) = B\llbracket \bar{t} / \overline{X} \rrbracket$$

where, and this will be the tricky part, B and $\overline{X}$ are *independent* of the input, $\bar{t}$. So:

$$\begin{array}{rl} & (\texttt{mkTerm}(o, n, a) \circ F)(\bar{t}) = \\ & = \texttt{mkTerm}(o, n, a, F(\bar{t})) \\ {\scriptstyle (\texttt{mkTerm}\ \text{def.})} & = \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}\lfloor, F(\bar{t})(i)(\overline{x_i})\lfloor))\lceil \\ {\scriptstyle (F\text{'s fact})} & = \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}\lfloor, c_i\llbracket \bar{t}, \overline{x_i} / \overline{u_i}, \overline{v_i} \rrbracket\lfloor))\lceil \\ {\scriptstyle (\star 7)} & = \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}\lfloor, c_i\llbracket \overline{s}, \overline{x_i} / \overline{u_i}, \overline{v_i} \rrbracket\llbracket \bar{t} / \overline{s} \rrbracket\lfloor))\lceil \\ {\scriptstyle (\cdot\llbracket\cdot/\cdot\rrbracket\ \text{def.})} & = \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}\lfloor, c_i\llbracket \overline{s}, \overline{x_i} / \overline{u_i}, \overline{v_i} \rrbracket\lfloor[\bar{t}\lfloor / \overline{s}\lfloor]\lceil\lfloor))\lceil \\ {\scriptstyle (\star 2)} & = \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}\lfloor, c_i\llbracket \overline{s}, \overline{x_i} / \overline{u_i}, \overline{v_i} \rrbracket\lfloor[\bar{t}\lfloor / \overline{s}\lfloor]))\lceil \\ {\scriptstyle (\star 10,\ \text{see below})} & = \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}\lfloor, c_i\llbracket \overline{s}, \overline{x_i} / \overline{u_i}, \overline{v_i} \rrbracket\lfloor))[\bar{t}\lfloor / \overline{s}\lfloor]\lceil \\ {\scriptstyle (\star 2)} & = \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}\lfloor, c_i\llbracket \overline{s}, \overline{x_i} / \overline{u_i}, \overline{v_i} \rrbracket\lfloor))\lceil\lfloor[\bar{t}\lfloor / \overline{s}\lfloor]\lceil \\ {\scriptstyle (\cdot\llbracket\cdot/\cdot\rrbracket\ \text{def.})} & = \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}\lfloor, c_i\llbracket \overline{s}, \overline{x_i} / \overline{u_i}, \overline{v_i} \rrbracket\lfloor))\lceil\llbracket \bar{t} / \overline{s} \rrbracket \end{array}$$

In the above, making sure $\star 10$ applies needs some care. Assume that for some j, l, the variable $s_j\lfloor$ is free in the lth body, which is $c_l\llbracket \overline{s}, \overline{x_l} / \overline{u_l}, \overline{v_l} \rrbracket\lfloor$. We need to make sure in this case that $\overline{x_l}\lfloor$ is not free in either $t_j\lfloor$ or $s_j\lfloor$. The latter is trivial by the choice of $\overline{s}$ (and holds for all indexes), but the former is not obvious. What we do know about $\overline{x_l}\lfloor$ is its definition:

$$\overline{x_l}\lfloor = \mathrm{newvar}_{a_l}(F(\bar{t})(l))\lfloor = \mathrm{newvar}_{a_l}(c_l\llbracket \bar{t}, 0^{a_l} / \overline{u_l}, \overline{v_l} \rrbracket)\lfloor$$

[4] Note that this special form of λ could be avoided if the fourth input type to `mkTerm` would take the terms first and then the index (instead of the $\mathsf{SubstFunc}_{a_i}$), but that would require a special composition operation instead.

but since $s_j\lfloor$ is free in $c_l\llbracket\overline{s}, \overline{x_l}/\overline{u_l}, \overline{v_l}\rrbracket\lfloor$, then $u_{l,j}\lfloor$ must appear in $c_l\lfloor$; therefore, the choice of $\overline{x_l}\lfloor$ above must pick variables that do not appear in $\overline{t_j}\lfloor$ so we're safe.

Going back to the main proof, the last term of the equality chain built so far was:

$$\texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}\lfloor, c_i\llbracket\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}\rrbracket\lfloor))\lceil\llbracket\overline{t}/\overline{s}\rrbracket$$

which has the $B\llbracket\overline{t}/\overline{X}\rrbracket$ structure that we're looking for, but we're not finished because both the B and the $\overline{X}$ parts depend on $\overline{t}$ — $\overline{x_i}$ is defined in terms of $\overline{t}$, $\overline{s}$ is defined in terms of $\overline{x_i}$, and both B and $\overline{X}$ parts contain instances of $\overline{s}$ (and B actually contains $\overline{x_i}$ as well).

So we choose $\overline{t}$-independent values now: let $\overline{x'_i} = \text{newvar}_{a_i}(c_i)$ and let $\overline{s'} = \text{newvar}_k(\overline{c}, \overline{x'_1}, \ldots, \overline{x'_k})$. we also need to show that in the above, using $\overline{x'_i}, \overline{s'}$ instead of $\overline{x_i}, \overline{s}$ is still the same value. In an attempt to simplify this we now choose n sets of variables $\overline{z_1} \in \textsf{Term}^{a_1}, \ldots, \overline{z_n} \in \textsf{Term}^{a_n}$, which are completely fresh: they do not appear in anything mentioned so far, including $\overline{t}$.

Now, back to our equality chain which left off at:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}\lfloor, c_i\llbracket\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}\rrbracket\lfloor))\lceil\llbracket\overline{t}/\overline{s}\rrbracket$$

By $\star$11:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\lfloor, c_i\llbracket\overline{s}, \overline{z_i}/\overline{u_i}, \overline{v_i}\rrbracket\lfloor))\lceil\llbracket\overline{t}/\overline{s}\rrbracket$$

Next, we use substitution to get $\overline{s'}$ inside - $\overline{s}$ are distinct, $\overline{s'}$ are distinct, and $\overline{s'}$ does not occur in $\overline{z_i}$:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\lfloor, c_i\llbracket\overline{s'}\llbracket\overline{s}/\overline{s'}\rrbracket, \overline{z_i}\llbracket\overline{s}/\overline{s'}\rrbracket/\overline{u_i}, \overline{v_i}\rrbracket\lfloor))\lceil\llbracket\overline{t}/\overline{s}\rrbracket$$

Because $\overline{s'}$ does not occur free in c_i, this would be the expansion of the following substitution by $\star$9:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\lfloor, c_i\llbracket\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}\rrbracket\llbracket\overline{s}/\overline{s'}\rrbracket\lfloor))\lceil\llbracket\overline{t}/\overline{s}\rrbracket$$

Combining $\cdot\llbracket\cdot/\cdot\rrbracket$ and $\star$2 we get:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\lfloor, c_i\llbracket\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}\rrbracket\lfloor[\overline{s}\lfloor/\overline{s'}\lfloor]))\lceil\llbracket\overline{t}/\overline{s}\rrbracket$$

$\overline{z_i}$ do not occur in either $\overline{s}$ or $\overline{s'}$ so we can use $\star$10:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\lfloor, c_i\llbracket\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}\rrbracket\lfloor))[\overline{s}\lfloor/\overline{s'}\lfloor]\lceil\llbracket\overline{t}/\overline{s}\rrbracket$$

Again, using $\cdot\llbracket\cdot/\cdot\rrbracket$ and $\star$2:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\lfloor, c_i\llbracket\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}\rrbracket\lfloor))\lceil\llbracket\overline{s}/\overline{s'}\rrbracket\llbracket\overline{t}/\overline{s}\rrbracket$$

Now, $\overline{s}$ does not appear in the `mkCTerm` except possibly for $\overline{s'}$ (because we know it is not in c_i or $\overline{z_i}$), so using $\star$9 we get:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\lfloor, c_i\llbracket\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}\rrbracket\lfloor))\lceil\llbracket\overline{s}\llbracket\overline{t}/\overline{s}\rrbracket/\overline{s'}\rrbracket$$
$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\lfloor, c_i\llbracket\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}\rrbracket\lfloor))\lceil\llbracket\overline{t}/\overline{s'}\rrbracket$$

Finally, using $\star$11 we get:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x'_i}\lfloor, c_i\llbracket\overline{s'}, \overline{x'_i}/\overline{u_i}, \overline{v_i}\rrbracket\lfloor))\lceil\llbracket\overline{t}/\overline{s'}\rrbracket$$

Our final term has the desired $B\llbracket\overline{t}/\overline{X}\rrbracket$ form, and now the B and the $\overline{X}$ parts are independent of $\overline{t}$. This is because:

- $\overline{x'_i}$ depends only on c_i;
- $\overline{s'}$ depends only on $\overline{x'_i}$ and $\overline{c}$, and therefore only on $\overline{c}$;
- and $\overline{u_i}$ and $\overline{v_i}$, just like $\overline{c}$, were derived from the assumption that the inputs are substitution functions.

QED.

7 Conclusions

The construction of the Term type was done to facilitate exposing internal Nuprl functionality to Nuprl users, which, it is hoped, will lead to a lightweight reflection implementation. We have shown the plausibility of basing logical reflection on higher-order abstract syntax, where each syntactic operator is denoted directly by another operator.

We are continuing the implementation of reflection in the Nuprl system along these lines, and hope to soon test this conjecture. The core rules reflecting syntax that we showed correct here, are already implemented, reusing existing internal functionality, without involving concrete syntax. Initial examples have indicated that it is, in fact, useful.

Acknowledgments

Thanks to Robert Constable for numerous discussions on these and closely related topics.

References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, Philadelphia, Pennsylvania*, pages 95–105, Los Alamitos, California, June 1990. IEEE Computer Press Society.
2. Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language.* PhD thesis, Cornell University, 1987.
3. Eli Barzilay. *Implementing Reflection in Nuprl.* PhD thesis, Cornell University, to appear in 2002.
4. R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.
5. Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems.* Dover Publications, New York, 1992.
6. Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised 5 report on the algorithmic language scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.
7. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.
8. Andrew M. Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction*, pages 230–255, 2000.

DOVE: a Graphical Tool for the Analysis and Evaluation of Critical Systems
(Extended Abstract)

Tony Cant, Jim McCarthy, and Brendan Mahony

Defence Science and Technology Organisation,
Department of Defence,
PO Box 1500, Edinburgh, South Australia 5111,
Australia

DSTO's DOVE project [2] aims to provide easy-to-use tools for the analysis and evaluation of critical defence systems. The current DOVE tool provides a graphical front-end to a state-machine reasoning environment developed in the Isabelle/HOL [4] environment.

- It allows the user to interact with a design in a highly visual way, performing many design, animation and proof activities through direct manipulation of a graphical presentation of the state-machine topology.
- The XIsabelle component of DOVE provides a graphical environment for managing the construction and execution of tactical proof-scripts.
- It provides the user with a comprehensive database of the state-machine design, including definitions of all the transitions, variables, constants, and properties associated with the design.
- It allows the user to generate a high-quality presentation of the design in PDF format via the LaTeX document preparation system [3].

Experience in the use of DOVE has suggested a number of possible approaches to improving the DOVE tool.

- Many critical systems involve complex interactions between numerous components. DOVE would benefit greatly from support for the hierarchical decomposition of designs and analysis activities.
- Many critical systems involve analog components, concurrency, stochastic and real-time interactions. An ability to treat such issues would greatly enhance the scope of the tool.
- High assurance development standards require the generation of numerous documents describing various formal analyses of a design. Being able to generate a number of different documents from a single formal design, in various formats and for various audiences would be of great value to DOVE users.
- Providing true assurance in a design involves the effective communication of the results of the analysis process to a human audience. This is a particular challenge for formal proofs in general and is nearly impossible where proofs are generated through the application of tactical programs as is the case in the current version of DOVE. In addition tactical proofs are extremely brittle and difficult to maintain. Proofs need to be structured in a manner more natural to the human reader and at a level of detail which ensures the communication of the salient points without overwhelming the user with arcane points of formal logic. If possible proofs should also benefit from graphical comprehension aids.
- The unstructured input of formal mathematical text can be a painstaking process even for the most experienced of users. An ability to assist and direct the construction of formulae is critical to the general adoption of a formal analysis tool. A related point is the need to provide screen support for the many and varied mathematical symbols and notations demanded by mathematically advanced users.
- As a formal model grows, the problems associated with any large data set become apparent. The user spends increasing amounts of time and effort searching for the information necessary to make progress. Tool support for the management of the design database is critical to the efficient performance of formal analysis techniques.

Concurrently with the development of the DOVE tool, important improvements have been made to the Isabelle environment itself.

- The Isar environment [6] now provides a literate and interactive environment for theory and proof development and presentation. Isar proof texts represent a significant improvement over tactical scripts, both in terms comprehensibility and in terms of maintainability. However, as Isar proofs now contain the statement of many intermediate facts, the need for tool support in the management of design and proof elements is once again made apparent.
- The introduction of the Isatool tool [7] has provided Isabelle with a mechanism to structuring formal developments as sessions and also for assisting in the preparation of a (single) high-quality document presentation of a session. However, the need to generate many views and presentations of a single session has not been addressed.
- Integration with the Emacs-based Proof-General [1] tool now offers tool-supported management of interactive theory-script execution. Unfortunately, the Proof-General environment ignores the hooks for tool support provided by Isabelle mechanisms such as the session concept, document generation, or importantly the assume/show presentations of proof goals which could easily be auto-inserted into the theory script.
- Support for the screen and document presentation of a range of mathematical symbols has been provided through integration with the X-Symbols[5] package. Unfortunately, this does little to address a fundamental problem with Isabelle, namely the strong coupling of formal syntax and mathematical presentation. An illustrative example of this problem is the inability to make use of the letter 'o' as a logical variable. The character has been co-opted to serve as a formal syntax for function composition. Such syntactic conflicts become more and more common as the size and complexity of the theory database increases.

With all of these factors and opportunities in mind, the DOVE project team has developed a fundamental re-design of the DOVE tool. We have identified the following priorities for the next generation of the DOVE tool.

- The basic function of the tool is to be the preparation of design assurance documents for the evaluation and certification of critical systems. It should allow the user to construct any number of documents in essentially arbitrary formats from a single project database.
- The tool should allow the user to group and maintain all formal elements, indexes and documents related to a given project within a single project artifact.
- The tool should include a structured, syntax-directed editing mode for formal mathematical input. This mode should allow the user to construct mathematical terms by making selections from a palette of available operators.
- The tool should offer a decoupling of formal presentation and formal syntactic layers. This allows individual users the maximum of presentation freedom without modification of the formal syntactic layer, an especially inconvenient process for library and predefined theories. The user should be able to craft presentations using arbitrary user supplied character fonts.
- The tool should provide powerful indexes and structured views of the modeling database. This is critical for the efficient use of the structured editor and should act as an enabling technology for powerful tool-support for a number of other user activities.
- The architecture of the tool should enable the future extension of the tool with specialised modeling or reasoning environments.
- The tool should include a reasoning environment for the analysis of hierarchies of processes, allowing the treatment of concurrency, real-time properties, and analog system components.
- The specification of temporal properties should enjoy a graphical presentation and importantly the proof of temporal properties for state machines should be performed as much as possible through interactions with the graphical presentations of the temporal properties and/or the state machine itself.

The next generation DOVE tool is currently in an advanced stage of design and initial prototype coding has begun on the following features.

- An XML document style has been developed to encompass all elements of a design project, including formal mathematics, indexing, and multiple document views.
- The graphical user interface and modeling database is being developed in Java. The use of an object-oriented programming language is expected to greatly simplify future extensions to the tool.

- The basis of the tool is the design editor, which allows the user to interact with the formal design using a familiar document construction metaphor. The design editor is based on the the Swing Document class, allowing it to be prototyped rapidly with an acceptable level of on-screen document presentation.
- The design editor will also control user-interaction with the underlying Isabelle/Isar environment. The editor automatically generates legal Isabelle theory files from the design document and controls the interactive processing of these files. Control of theory processing will be maintained through a 'processed-up-to' marker which is additional to the usual text entry cursor. The user controls theory processing by moving this logic cursor around the document. The design editor mediates with Isabelle/Isar, providing feedback to the user on the results of processing, particularly of errors encountered.
- Printed document presentation will be provided through the use of the LaTeX document preparation system. Access to more sophisticated document construction features, such as precision mathematical typesetting and bibliographies, will be provided by allowing the verbatim insertion of LaTeX code into the generated LaTeX output.
- A structured mathematical editor is being prototyped with sophisticated support for the use of mathematical fonts in the Truetype format. Mathematical input proceeds through the selection of constants, type constructors, syntactic operators, *et cetera* from palettes that list the defined elements from the appropriate class.
- A symbolic presentation layer has been incorporated into the structured editor, allowing the user to change the presentation of mathematical operators with a minimum of impact on the generated Isabelle theories.
- Sophisticated tools for the description of hierarchies of components have been developed in Isabelle, inspired by the Z schema calculus and mathematical toolkit.
- Isabelle support for the design and analysis of networks of input/output process has been developed. Dataflow-style diagrams will be used to visualisation these network designs and to provide a graphical user-interface for their construction and manipulate.
- Enhanced Isabelle support for real-analysis is being developed in collaboration with the Software Verification Research Centre (SVRC) at the University of Queensland.
- Initial investigations into support for probabilistic programming techniques are being carried out in collaboration with Macquarie University.

References

1. D Aspinall. Proof General. http://www.proofgeneral.org/.
2. A Cant, J McCarthy, and B Mahony. DOVE: Design Oriented Verification Environment. http://www.dsto.defence.gov.au/esrl/itd/dove/index.html.
3. L Lamport. *LaTeX: A Document Preparation System*. Addison Wesley, 2^{nd} edition, 1994.
4. T Nipkow, L Paulson, and M Wenzel. Isabelle's Logics: HOL. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2002/doc/logics-HOL.pdf.
5. C Wedler. X-Symbols for WYSIWYG in Emacs. http://x-symbol.sourceforge.net/.
6. M Wenzel. The Isabelle/Isar Reference Manual. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2002/doc/isar-ref.pdf.
7. M Wenzel and S Berghofer. The Isabelle System Manual. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2002/doc/system.pdf.

Formalising General Correctness

Jeremy E. Dawson[1]*

Department of Computer Science
Australian National University, Canberra ACT 0200, Australia
jeremy@csl.anu.edu.au
http://csl.anu.edu.au/~jeremy/

Abstract. We consider the abstract command language of Dunne, and his account of general correctness. We provide an operational interpretation of his abstract commands, and use the automated theorem proving system Isabelle to prove that this operational interpretation leads to Dunne's semantics.

Keywords: general correctness, abstract commands.

1 Introduction

General correctness was introduced as an alternative to partial correctness and total correctness by Jacobs & Gries (1985) [5], see also Nelson (1989) [7]. Jacobs & Gries use a relational model, representing a program as a relation between initial states and final states: their space of final states includes $\perp$, representing non-termination. In this way they can distinguish when a program guarantees termination, guarantees non-termination, or neither. Neither partial correctness nor total correctness (alone) can do this.

In [1] and [2], Dunne gives an account of general correctness, in which he gives a set of "abstract commands", with associated semantics. For each abstract command, Dunne gives its semantics in terms of its termination condition, its weakest liberal precondition predicate and its *frame*, which is (loosely) the set of program variables which might be altered by the command. From these one can derive total-, partial- *and* general correctness semantics.

We describe the abstract commands in terms of an operational interpretation similar to that of Jacobs & Gries. We then use the automated prover Isabelle to show that this interpretation implies the semantics given by Dunne. We also use Isabelle to prove some of his more difficult results. This paper refers to results proved in Isabelle — the code is available via the author's home page (above).

In [3], Gordon provided an operational interpretation of programs (commands), and used the HOL theorem prover to verify the axioms (rules) of Hoare logic. He explains in detail certain problematic aspects of such work, which we will allude to briefly.

In [4], Harrison formalized Dijkstra's program logic in the HOL theorem prover, using a relation between states and outcomes to model commands.

2 Modelling Commands and Conditions

Commands Typically one models a command (or program) as a function acting on the machine state. A deterministic command which must terminate can be modelled as a function returning simply a single new machine state. A deterministic command which may or may not terminate could be modelled as a function which returns either a new state or nothing, representing the idea that a non-terminating command returns no result. However if we represent a non-deterministic program as a function which returns a set of new states, then this leaves us without a way of representing non-termination as one of several possible outcomes.

We also want to represent commands which are infeasible. (These are a useful building-block, even if you don't want to write such programs, as Dunne discusses). In fact this, rather than non-termination, is naturally represented by a command returning no new state.

The solution (Plotkin [8], also used by Harrison [4]) is to consider command *outcomes*, where an outcome is either termination in a new state or non-termination.

* Supported by an Australian Research Council Large Grant

Conditions Boolean expressions, or conditions, on the machine state, occur in work such as this in two contexts. Firstly, many commands (such as if then else, or while do) incorporate conditions on the state. A state is typically represented as a function from the set of variable names to their values. The condition in such a command will most naturally be represented as text in the programming language, or as an abstract syntax tree, but as it will be capable of being evaluated in any machine state, we might well think of it as a function of type *state* → *bool* (and then we could treat the notion of state as an abstract entity).

Secondly, a condition Q can appear in an expression $wlp(C, Q)$ (where wlp means weakest liberal pre-condition), or in $\{P\}C\{Q\}$ (Hoare logic). It may be most natural to think of these as predicates on states (or functions of type *state* → *bool*). However the rule for wlp, and a related rule in Hoare logic, are

$$wlp(x := E, Q) = Q[x := E] \qquad\qquad \{P[x := E]\}\ x := E\ \{Q\}$$

By $Q[x := E]$ we mean Q with occurrences of x replaced by E; various other authors write this as $Q[x/E]$, $Q[E/x]$, $Q_{E\to x}$, $Q\langle E/x\rangle$, $\{E/x\}Q$, with, confusingly, both the first two being popular. The notion of substitution in these rules is meaningless when P and Q are arbitrary predicates on states: they require P and Q to be expressions written in the command language, or something like it, or as, say, abstract syntax trees, containing literal program variable names. Note that the language for these predicates must not be able to express a condition like "no two different variables may have the same value" (for, then, what would $Q[x := E]$ mean?) However, P and Q *may* also contain logical variables, as in the following Hoare logic example (taken from Gordon[3, §5.0], where X, Y, Z denote program variables and x, y denote logical variables)

$$\{X = x \wedge Y = y\}Z := X; X := Y; Y := Z\{X = y \wedge Y = x\}$$

It is also worth noting at this point that where boolean expressions are used in abstract commands (such as the guarded command $P \to A$ and the preconditioned command $P|A$) the boolean P is not treated as a fragment of code but rather as an arbitrary predicate on the state. Thus, as is clear from Dunne's treatment of these commands, the possibility of P looping or producing other than a single answer is not considered.

Gordon [3] discusses these issues. What this means for us now is that our analysis of many commands (*not* including assignment) can be performed at the level of abstraction where a boolean expression is modelled as a predicate on states, and a command is modelled as a function from states to sets of outcomes. The next section contains the analysis at that level.

Frames Dunne has also defined that each abstract command has a *frame*. Loosely, this is the set of variables which "might" be affected. Note, however, that $frame(x := x) = \{x\}$. Also, from any command a new command may be defined which has an enlarged frame but is otherwise the same.

Stating the frame of a command does not contribute to a description of what the command does, so we can show, for example, that two commands behave the same way, without considering their frames. The work in this section proceeds on this basis. Note that the results are therefore subject to the proviso that two abstract commands are in fact not the same if their frames differ. We think that the relevant proofs about frames would be quite straightforward.

Consideration of literal commands and expressions and of the frames of commands is deferred to the following section, as is that of the assignment command.

3 Commands as transformations of state

3.1 Monadic Types

As mentioned, we model a command as a function from states to sets of outcomes. Here is the formal definition of the type *outcome*.

```
datatype outcome = NonTerm | Term state
```

So when we model sequencing of two commands A and B, we first apply A to a given state, obtaining a set of outcomes, and we must then apply B, a function of type *state* → *outcome set*, to the set of outcomes obtained from A. We can think of this as "extending" the function B to a function *ext* B of type *outcome set* $\rightarrow$ *outcome set*. When this can be done in a way that satisfies certain conditions, we call the relationship between the types a "monad". See Wadler [9] for further information on monads.

In fact, this is an example of a *compound monad*. The type *outcome*, relative to the type *state*, is a monad, where the extension function, of type (*state* $\rightarrow$ *outcome*) $\rightarrow$ (*outcome* $\rightarrow$ *outcome*) would be given by

$$\begin{aligned} &\textit{exto}\ f\ \texttt{NonTerm} = \texttt{NonTerm} \\ &\textit{exto}\ f\ (\texttt{Term}\ s) = f s \end{aligned}$$

For any type α, the type α *set* (the type of sets of things of type α) is also a monad, where the extension function, of type ($\alpha \rightarrow \alpha$ *set*) $\rightarrow$ (α *set* $\rightarrow \alpha$ *set*), would be given by

$$\textit{exts}\ f\ os = \bigcup_{o \in os} f\ o$$

Apart from the extension function, specification of a monad includes a *unit* function, which converts a value of the "base" type, usually in a rather natural way, to a value of the monadic type. For the two monads mentioned, we have

$$\begin{aligned} &\textit{unito} : \textit{state} \rightarrow \textit{outcome} \qquad\qquad &&\textit{units} : \alpha \rightarrow \alpha\ \textit{set} \\ &\textit{unito}\ s = \texttt{Term}\ s &&\textit{units}\ e = \{e\} \end{aligned}$$

Note also that the extension function is often called *bind* and written in infix format (as in [9]), so *ext* f s = s *bind* f.

Two monads cannot in general be composed to form another monad, but the first monad mentioned above can in general be composed with any other monad to give a compound monad (see [6, §7.3]). The formulae for the extension function, both generally (in terms of *units* and *exts*) and for our specific choice of *units* and *exts*, are given below. In the specific case, *extos* has type (*state* $\rightarrow$ *outcome set*) $\rightarrow$ (*outcome set* $\rightarrow$ *outcome set*).

$$\begin{aligned} &\textit{extos}\ f\ os\ = &&\textit{extos}\ f\ os\ = \\ &\quad \textit{let}\ f'\ (\texttt{Term}\ s) = f\ s &&\quad \textit{let}\ f'\ (\texttt{Term}\ s) = f\ s \\ &\qquad f'\ \texttt{NonTerm} = \textit{units}\ \texttt{NonTerm} &&\qquad f'\ \texttt{NonTerm} = \{\texttt{NonTerm}\} \\ &\quad \textit{in}\ \textit{exts}\ f'\ os &&\quad \textit{in}\ \textstyle\bigcup_{o \in os} f'\ o \end{aligned}$$

As mentioned above, a monad consists of functions *unit* and *ext* (of appropriate types), which must satisfy certain conditions, as follows:

$$\begin{aligned} \textit{ext}\ k \circ \textit{unit} &= k && \text{(Left Unit)} \\ \textit{ext}\ \textit{unit} &= \textit{id} && \text{(Right Unit)} \\ \textit{ext}\ (\textit{ext}\ h \circ k) &= \textit{ext}\ h \circ \textit{ext}\ k && \text{(Assoc)} \end{aligned}$$

Let *seq* A B denote the sequencing of commands A and B (where A, B and *seq* A B are of type *state* $\rightarrow$ *outcome set*). As noted, we want to first apply A to the given state, obtaining a set of outcomes; we must then apply the extension of B (of type *outcome set* $\rightarrow$ *outcome set*) to that set of outcomes. That is, *seq* A B = *extos* $B \circ A$. Then we can prove the associativity of *seq* thus:

$$\begin{aligned} \textit{seq}\ A (\textit{seq}\ B\ C) &= \textit{extos}\ (\textit{seq}\ B\ C) \circ A \\ &= \textit{extos}\ (\textit{extos}\ C \circ B) \circ A \\ &= \textit{extos}\ C \circ \textit{extos}\ B \circ A \qquad \text{by the monad rule (Assoc)} \\ &= \textit{extos}\ C \circ \textit{seq}\ A\ B \\ &= \textit{seq}\ (\textit{seq}\ A\ B)\ C \end{aligned}$$

This is proved in Isabelle as **seq_assoc**. Dunne [1, §7] uses ';' for sequential composition, so he writes *seq* A B as $A; B$.

The *unit* function, of type *state* → *outcome set*, of the compound monad is given by

$$unitos\ s = units\ (\texttt{Term}\ s) = \{\texttt{Term}\ s\}$$

This represents the command *skip*, which always terminates in its initial state.

3.2 Refinement

As we will often just give Isabelle code, we mention some less obvious Isabelle notation. The "?" indicates a variable for which anything (of a suitable type) may be substituted. Some set and function notation (mathematical and Isabelle equivalents) follows:

$a \notin \bigcup_{b\in C} \cdot D$	`a ~: UN b:C. D`
$\{a\} \cup (C \cup D) \subseteq E \cap F \backslash G$	`insert a (C Un D) <= E Int F - G`
$\lambda x.E$	`(%x. E)`

We define functions corresponding to *wlp*, *trm*, and *wp* of [1, §2].

```
wlpm ?cm ?bm ?state == ALL nst. Term nst : ?cm ?state --> ?bm nst
trmm ?cm ?state      == NonTerm ~: ?cm ?state
wpm ?cm ?bm          == wlpm ?cm ?bm && trmm ?cm
```

Here `&&` and `||` lift conjunction and disjunction over states, and `--->` is the "is stronger" relation between predicates, so

```
?p ---> ?q      ==  ALL s. ?p s --> ?q s
(?p && ?q) ?s   ==  ?p ?s & ?q ?s
(?p || ?q) ?s   ==  ?p ?s | ?q ?s
```

These definitions work with commands and conditions as functions of type *state* → *outcome set* and *state* → *bool* respectively. We note that a command (as such a function) is uniquely determined by its *wlp* and termination conditions. This is proved in Isabelle as `unique`. Later we will introduce corresponding (differently named) functions which take abstract syntax trees as arguments.

In [1, §5] Dunne discusses several notions of refinement, including general-, total- and partial-correctness refinement. The second equivalent definition of `gencref` is derived from Dunne's (Gcref2) ([2, §2.1]).

```
totcref ?Am ?Bm   ==  ALL Qm. wpm ?Am Qm ---> wpm ?Bm Qm
partcref ?Am ?Bm ==  ALL Qm. wlpm ?Am Qm ---> wlpm ?Bm Qm
gencref ?Am ?Bm   ==  partcref ?Am ?Bm & totcref ?Am ?Bm"
gencref ?Am ?Bm   ==  partcref ?Am ?Bm & (trmm ?Am ---> trmm ?Bm)
```

From these definitions, we have derived more direct characterizations of these three notions of refinement. It is worth noting that the characterization for general correctness is simpler than the other two although it is defined in terms of both of them; this no doubt explains how general correctness semantics often seems simpler than either partial or total correctness semantics.

```
totcref ?Am ?Bm  =  (ALL st. ?Bm st <= ?Am st | NonTerm : ?Am st)
partcref ?Am ?Bm =  (ALL st. ?Bm st <= insert NonTerm (?Am st))
gencref ?Am ?Bm  =  (ALL state. ?Bm state <= ?Am state)
```

3.3 Meaning of Commands

skip, perhaps, magic, abort [1, §7] *skip* is the command which is feasible, terminates and does nothing to the state. It is exactly the function *unitos*. It follows immediately from the (Left Unit) and (Right Unit) monad laws that *skip* is an identity (left and right) for the binary function *seq*. These are proved in Isabelle as `seq_unitL` and `seq_unitR`. We define

```
perhaps ?st  ==  {Term ?st, NonTerm}
magic ?st    ==  {}
abort ?st    ==  {NonTerm}
```

preconditioned command [1, §7] The command $P|A$ is the same as A except that, if P does not hold, then $P|A$ *may* fail to terminate.

```
precon ?bm ?cm ?state == if ?bm ?state then ?cm ?state else insert NonTerm (?cm ?state)
```

guarded command [1, §7] The command $P \longrightarrow A$ is the same as A if P holds, but is *infeasible* (the outcome set is empty) if P does not hold.

```
guard ?bm ?cm ?state == if ?bm ?state then ?cm ?state else {}
```

A command has a "natural" guard and precondition. Here `fis` A means A is *feasible*, that is, its outcome set is non-empty. We have proved

```
fis_guard  =  "guard (fis ?Am) ?Am = ?Am"
pc_trm     =  "precon (trmm ?Am) ?Am = ?Am"
```

choice In [1, §7] Dunne defines a binary operator, $A \Box B$, for *bounded choice*: $A \Box B$ is a command which can choose between two commands A and B. This is a special case of choice among an arbitrary set of commands, defined by

$$choice\ C\ s = \bigcup_{c \in C} c\ s$$

```
choice ?cms ?state == UN cm:?cms. cm ?state
```

From these, we prove the definitions, and some other results, of Dunne.

```
perhaps_alt = "perhaps = precon (%st. False) unitos"
magic_alt = "magic = guard (%st. False) ?A"
abort_alt = "abort = precon (%st. False) (guard (%st. False) ?A)"
pma = "seq perhaps magic = abort"
asp = "choice {abort, unitos} = perhaps"
```

concert [1, §12] The command $A\#B$ represents A and B executing independently, on seperate copies of the state: whichever of A or B terminates first gives the effect of $A\#B$. Thus the possible outcomes of $A\#B$ are:

- `Term` s, if it is an outcome of A,
- `Term` s, if it is an outcome of B,
- `NonTerm`, if it is an outcome of *both* A and B.

```
conc ?Am ?Bm ?state == concrs (?Am ?state) (?Bm ?state)
concrs ?cr1 ?cr2 == ?cr1 Un ?cr2 - {NonTerm} Un {NonTerm} Int ?cr1 Int ?cr2
```

Interestingly, this means that if B is *magic* (everywhere infeasible), then $A\#B$ is just A with any possibility of non-termination removed (difficult though it is to see from the first sentence above!). This is proved in Isabelle as `conc_magic`.

The *wlp* and termination conditions for these commands, which are used by Dunne to *define* these commands, are proved in Isabelle from our definitions, as `precon_trm`, `precon_wlp`, `guard_trm`, `guard_wlp`, `seq_trm`, `seq_wlp`, `choice_trm`, `choice_wlp`, `conc_trm` and `conc_wlp`. Dunne's results Xpre, Xguard, Xassump and Xassert are also proved in Isabelle, under the same names.

3.4 Repetition and Iteration

finite repetition [1, §7] Dunne defines $A^0 = skip$ and $A^{n+1} = A; A^n$. A very convenient result which we proved, called `rep_Suc'`, is that $A^{n+1} = A^n; A$.

repetitive closure [1, §12] We also defined *repall* $c\ s = \bigcup_n rep\ n\ c\ s$, ie.

```
repall ?cm ?state == UN n. rep n ?cm ?state
```

that is, `repall` A is the (unbounded) choice of any number of repetitions of A. The termination condition for `repall` A is that for every n, A^n terminates (proved as `repall_term`).

The *repetitive closure* of A is A^*, where the outcomes of A^* are those of `repall`, augmented by `NonTerm` in the case where it is feasible to execute A infinitely many times sequentially (we call this an "infinite chain"). It is considerably easier to define this concept operationally than in terms of *wlp* and *trm*. The definition of this circumstance asserts an infinite sequence of states, of which each is reachable from the previous one. We omit the Isabelle definition.

$$infch\ A\ s \equiv \exists f.\ f\ 0 = s \wedge (\forall n.\ \texttt{Term}\ (f\ (n+1)) \in A\ (f\ n))$$

Thus we have the definition

```
repstar ?cm ?state == repall ?cm ?state Un (if infch ?cm ?state then {NonTerm} else {})
```

It may be noted that in [1, §10], Dunne defined a predicate *cic* ("cycles and infinite chains"), with the intended meaning (in effect) that *cic* $A\ s$ be true if A, executed in state s, might not terminate. However the definition made *cic* $A\ s$ true in the situation where A could be executed any given n times sequentially, which is *not* sufficient to ensure an infinite chain of executions. (It would be sufficient under an assumption of bounded non-determinacy, see [4, §3]). As is a common experience, we did not discover this discrepancy until trying to perform Isabelle proofs based on the definition in question.

We have proved some useful results, such as

`wlpca` : $wlp(A^*) = wlp(\texttt{repall}A)$ (since they differ only in that A^* has an additional possibility of non-termination)

`seq_repstar` : $A^*; A = A; A^*$

In [1, §12] Dunne mentions that repetitive closure could be defined using *Egli-Milner* approximation [1, §6].

$$A \leq_{em} A' \equiv A \sqsubseteq_{tot} A' \wedge A' \sqsubseteq_{par} A$$

where $\sqsubseteq_{tot}$ and $\sqsubseteq_{par}$ denote respectively total- and partial-correctness refinement. Then A^* is a least fixpoint under the ordering $\leq_{em}$:

$$A^* \equiv \mu_{em} X.(A; X) \Box skip$$

We show in Isabelle that our definition of A^* implies this result. Here `fprep_alt2` is a paraphrase of our definition of `fprep` (`fprep` $A\ X$ means $X = (A; X)\Box skip$), `repstar_isfp` says that A^* is a fixpoint, and `repstar_is_lfp` says that A^* is less than or equal to, in the Egli-Milner ordering, any given fixpoint Y.

```
fprep_alt2 = "fprep ?Am ?Xm = (?Xm = choice {seq ?Am ?Xm, unitos})"
repstar_isfp = "fprep ?Am (repstar ?Am)"
repstar_is_lfp = "fprep ?Am ?Ym ==> egMil (repstar ?Am) ?Ym"
```

Dunne (personal communication) also defines $trm(A^*)$ and $wlp(A^*, Q)$ as fixpoints:

$$trm(A^*) = \nu Y.wp(A, Y)$$
$$wlp(A^*, Q) = \mu Y.wlp(A, Y) \wedge Q$$

where μ and ν denote the least and greatest fixpoints, that is the weakest and strongest (respectively) fixpoints. We also prove these results in Isabelle, based on our definition of A^*. `trfp` and `wrfp` say that $trm(A^*)$ and $wlp(A^*, Q)$ are fixpoints of the respective functions. `trsfp` says that $trm(A^*)$ is equal or weaker than any given fixpoint Y, and similarly for `wrwfp`.

```
trfp = "let trmstar = trmm (repstar ?Am) in trmstar = wpm ?Am trmstar"
trsfp = "?Y = wpm ?Am ?Y ==> trmm (repstar ?Am) ---> ?Y"
wrfp = "let wlpstar = wlpm (repstar ?Am) ?Qm in wlpstar = (wlpm ?Am wlpstar && ?Qm)"
wrwfp = "?Y = (wlpm ?Am ?Y && ?Qm) ==> ?Y ---> wlpm (repstar ?Am) ?Qm"
```

3.5 Monotonicity

For developing a program by starting with an abstract expression (of requirements), and progressively refining it to a concrete program, it is important that the abstract commands constructors are monotonic with respect to general-correctness refinement ($\sqsubseteq_{gen}$).

Given our characterization of $A \sqsubseteq_{gen} B$ as ($\forall state.\ B\ state \subseteq A\ state$), and our operational definition of commands in terms of their outcome sets, it is easy to see that all the constructors mentioned are monotonic. In any event, they are proved in Isabelle as (for example) `seq_ref_mono`, `rep_ref_mono`, `repstar_ref_mono`.

3.6 The *while* loop

In [1, §7, §12] Dunne defines

$$\begin{aligned} \textit{if } G \textit{ then } A \textit{ end} &\equiv (G \to A) \Box (\neg G \to \textit{skip}) \\ \textit{while } G \textit{ do } A \textit{ end} &\equiv (G \to A)^*; \neg G \to \textit{skip} \end{aligned}$$

The definition of *while* which is intuitive to programmers is

$$\textit{while } G \textit{ do } A \textit{ end} \equiv \textit{if } G \textit{ then } A; \textit{while } G \textit{ do } A \textit{ end end}$$

We cannot use this as a definition in Isabelle since it is recursive as it stands it is non-terminating, and when applied to a particular state, may not terminate. So in Isabelle we have defined *while* as does Dunne, and have proved that it satisfies the "intuitive" definition.

```
while_prog = "while ?G ?A = ifthen ?G (seq ?A (while ?G ?A))"
```

4 Frames and Variable Names

In §3, we viewed a command as a function from a state to a set of outcomes, and a condition as a predicate on states. In this treatment, the view of a state was abstract. As discussed in §2, there are various ways in which a full treatment needs to be more concrete, namely

- referring to program variables
- having conditions in a form in which we can substitute for a program variable
- specifying a frame for a command

In this section we discuss those abstract command constructors which require us to address these issues.

In our Isabelle model, the program variable names are strings and they take natural number values. As a state is an assignment of variables to values, we have the type definition `state = "string => nat"`

indeterminate assignment [1, §12] Where x is a (list of) variables, and P is a predicate, the command $x : P$ assigns values to the variable(s) in x in any way such that the change of state satisfies P. More precisely, if α is the "current alphabet" (the set of variables whose names are currently "in scope"), and x_0 is the set of variable names in x, but with subscript 0 added, then P is a predicate on $\alpha \cup x_0$. (The paper [1] says $\alpha \cup \alpha_0$ – we comment on this below). The subscripted variable names represent the values of those variables before the command is executed. We model such a P as a function on two states, so our definition of this command is

```
indetass ?vars ?P ?s == Term ` (Collect (?P ?s) Int chst ?vars ?s)
```

where `chst ?vars ?s` means the set of states which differ from `?s` only in the variables `?vars`, `` f ` X `` means $\{f\ x \mid x \in X\}$, and `Collect (?P ?s)` means $\{s' \mid P\ s\ s'\}$.

prd [1, §10] The "before-after" predicate *prd* specifies conditions under which the command *may* terminate in a state where variables have certain given values. Dunne defines this as

$$prd\ (A) \equiv \neg wlp\ (A, x \neq x')$$

where x' are new (logical) variables corresponding to the program variables. We define **prds** and **prdm**, as

```
prds ?strs ?dashed ?Am == Not o wlpm ?Am (%st. EX str:?strs. st str ~= ?dashed str)
prdm ?dashed ?Am == Not o wlpm ?Am (%st. st ~= ?dashed)
```

where ?dashed, of type *state*, represents the values x', and **prdm** is a simpler version of **prds** for use when x can be taken to be all variable names. As a sort of inverse to this definition, Dunne gives $wlp\ (A, Q) = \forall x'. prd\ (A) \Rightarrow Q[x := x']$ which we prove as

```
wlp_prd = "wlpm ?Am ?Qm ?state = (ALL dashed. prdm dashed ?Am ?state --> ?Qm dashed)"
```

In [2, §9] Dunne states the result $prd\ (x : P) = P[x_0, x := x, x']$. We proved a corresponding result for the special case where x represents all variable names

```
indetass_prd = "prdm ?dashed (indetass UNIV ?P) ?state = ?P ?state ?dashed"
```

but found that we could not prove the stated result generally. It turned out that Dunne's result requires that P be a predicate on $\alpha \cup x_0$, not on $\alpha \cup \alpha_0$ (as stated in the paper). This is another example of the common situation that attempting to prove such results formally detects points such as this which can easily be overlooked in an informal treatment.

unbounded choice [1, §7] The command $(@z.A)$ means that variable z is to be set to any value and then A is to be executed. But z is to be a "local" variable in A; if, for example, Q contains z, then it is a *different* z from that in A. In other words, the notation correctly reflects that z behaves as normal for a bound variable (it can be α-converted with no change in meaning).

So we model this command as follows:

- set variables z to arbitrary values
- execute A
- reset variables z to their initial values

```
setstrs ?strs ?strfun ?state ?str == if ?str : ?strs then ?strfun ?str else ?state ?str
revert ?strs ?Am ?initst == mapos (setstrs ?strs ?initst) (?Am ?initst)
at ?strs ?Am ?initst ==
  let initptf = %strfun. setstrs ?strs strfun ?initst;
     initptc = %x. UNION UNIV (?Am o initptf)
  in revert ?strs initptc ?initst
```

Here, UNION UNIV $F = \bigcup_x F\ x$, and mapos is the monadic "map" function:

$$\begin{aligned} mapos\ f\ ocset &= \{mapo\ f\ s \mid s \in ocset\} \\ mapo\ f\ (\mathtt{Term}\ s) &= \mathtt{Term}\ (f\ s) \\ mapo\ f\ \mathtt{NonTerm} &= \mathtt{NonTerm} \end{aligned}$$

We then proved

```
at_trm = "trmm (at ?strs ?Am) = allstrs ?strs (trmm ?Am)"
```

where allstrs strs B s means that for any other state s' obtained by taking s and setting the variables strs to any values, B s' holds. We tried to prove

```
wlpm (at ?strs ?Am) ?Qm = allstrs ?strs (wlpm ?Am ?Qm)
```

but could not. This reflected the fact that the formula for *wlp* ($@z.A$) given by Dunne assumes that Q does not involve z. (As noted above, the α-convertibility of z in $@z.A$ means that we can sensibly assume this). In fact we proved

```
at_wlp = "indep ?strs ?Qm ==> wlpm (at ?strs ?Am) ?Qm = allstrs ?strs (wlpm ?Am ?Qm)
```

where `indep` z Q means that Q is "independent" of z. As Q is a semantic expression, not a syntactic one (see §4.1), "independent" was defined to mean that changing z does not change Q.

4.1 Assignment; the Syntactic View

As noted in §2, $wlp(x := E, Q) = Q[x := E]$, which is only meaningful when Q is some structure in which we can define substitutions. So we have defined types for the abstract-syntax-tree version of integer and boolean expressions, thus (abbreviated):

```
datatype exp = Num nat
             | Var string
             | Pluss exp exp
             | Minus exp exp
             | Timess exp exp

datatype bexp = Eq exp exp
              | Lt exp exp
              | Le exp exp
              | Gt exp exp
              | Ge exp exp
              | Nott bexp
              | T
              | F
              | And bexp bexp
              | Or bexp bexp
              | Imp bexp bexp
```

We defined substitution functions, of the following types

```
  expSub   :: "string => exp => exp => exp"
  bexpSub  :: "string => exp => bexp => bexp"
```

where (for example) `expSub` x E M means $M[x := E]$. We also defined functions to translate an expression (type *exp* or *bexp* which we will call a **syntactic** expression) to the corresponding function of type *state* $\rightarrow$ *nat* or *state* $\rightarrow$ *bool* (which we will call a **semantic** expression). We may also say the semantic expression is the "meaning" of the syntactic expression. Obviously, distinct syntactic expressions may have the same meaning, and therefore the "=" symbol in a proposition of the form "*wlp* $(A, Q) = \ldots$" can only be sensibly interpreted as equality of semantic expressions, notwithstanding that in "$wlp(x := E, Q) = Q[x := E]$", the right-hand side is only meaningful as a syntactic expression. We can talk about syntactic and semantic *commands* also.

```
types
  expMeaning = "state => nat"
  bexpMeaning = "state => bool"

consts
  expMng   :: "exp => expMeaning"
  bexpMng  :: "bexp => bexpMeaning"
```

We can then prove the following results, and corresponding ones for boolean expressions.

```
subLemma = "expMng (expSub ?x ?E ?Q) ?state = expMng ?Q (?state(?x := expMng ?E ?state))"
sub_equiv = "expMng ?Q = expMng ?R --> expMng (expSub ?x ?E ?Q) = expMng (expSub ?x ?E ?R)"
```

Here $f(x := E)$ is Isabelle notation for the function that is like f except that its value at argument x is E. The first of these results relates substitution for a variable in an expression to assignment to that variable in the state. The second expresses that if two syntactic expressions have the same meaning, then the results of making the same substitution in the two of them also have the same meaning. (Thanks to Dunne for pointing out the need for this result).

We are now in a position to define assignment and prove its properties. We define **assignv** and **assigne** for the assignment, to a variable, of a value and a (semantic) expression respectively. We also define **assignvs** for the assignment of values to a set of variables.

```
assignv ?var ?n ?state == {Term (?state(?var := ?n))}
assigne ?var ?E ?state == assignv ?var (?E ?state) ?state
assignvs ?strs ?strfun ?state == {Term (setstrs ?strs ?strfun ?state)}
```

We can then prove ass_trm (which is trivial an assignment terminates), and ass_wlp, which says $wlp(x := E, Q) = Q[x := E]$.

```
ass_wlp = "wlpm (assigne ?x (expMng ?E)) (bexpMng ?Q) = bexpMng (bexpSub ?x ?E ?Q)"
```

4.2 Normal Form

In [2, §7.1] Dunne gives the following result, giving a "normal form" for an abstract command A.

$$A = trm\ (A)\ |\ @x'.prd\ (A) \rightarrow x := x'$$

Here x is the frame of A (which we first take to be the entire current alphabet of variable names), and x' is a corresponding set of logical variables, with names dashed. For this purpose we want somewhat different definitions of @ and of A, involving a set of logical variables x', one for each program variable. So we use a function dashed, of type *state*, which gives the values of these logical variables.

```
atd ?Ad ?state == UN dashed. ?Ad dashed ?state
```

Here ?Ad is not a semantic command, but a function which, given a "dashed" state as argument, returns a semantic command. Then also the assignment $x := x'$ (where x represents *all* variables) becomes the replacing of state x by "state" x'. Thus we prove the following corresponding result.

```
ACNF = "?A = precon (trmm ?A)
   (atd (%dashed. guard (prdm dashed ?A) (%st. {Term dashed})))"
```

We also proved a corresponding result for the case where x is a proper subset of all variables. Here Dunne's result requires that A does not change variables outside the set x. Rather than specify this requirement as such, we proved a result whose left-hand-side means "A restricted to x", that is, as though you executed A and then reset the variables outside x to their original values.

```
ACNFs = "revert (- ?x) ?A = precon (trmm ?A)
   (atd (%dashed. guard (prds ?x dashed ?A) (assignvs ?x dashed)))"
```

4.3 Frames

In Dunne's formulation [1, §7], each abstract command comes decorated with a frame, and the frame of the new command is defined individually for each abstract command constructor: for example

$$frame\ (A \Box B) = frame\ (A \# B) = frame(A) \cup frame(B)$$

However we are unable to give an exact semantic meaning to the frame in a similar sense to the meaning we have given to commands so far. The frame may be thought of as a set of variables "potentially" set by the commands, but it can be larger than the set of variables actually set by the command. The frame may be smaller than the set of variables read by the command, and two commands which have the same semantic meaning can have different frames. Accordingly we could not attempt to prove the statements about frames given by Dunne in the definitions of abstract commands from our operational model, in the way we have done for their *wlp* and *trm* conditions. The best one could do is to attempt to prove that for any abstract command the frame of the result *contains* the set of variables which are changed by the command. However this does not look at all difficult in any case, and so we have not included frames in our model.

parallel composition [1, §12] This is the only abstract command operator whose meaning depends on the frames of its operands. The command $A||B$ executes A and B, independently, each on its own copy of the variables in its frame, and waits until both have terminated. (Thus, non-termination is a possible outcome of $A||B$ if it is possible for either A or B). We say a new state resulting from A is *compatible* with a new state resulting from B if these new states agree on the values they give to the variables in $\mathit{frame}(A) \cap \mathit{frame}(B)$. Then, for each (s_A, s_B), where s_A and s_B are compatible new states resulting from A and B respectively, there is an outcome `Term` s_{AB} of $A||B$, where s_{AB} is given by:

- the new values of variables in $\mathit{frame}(A) \cap \mathit{frame}(B)$ are as in s_A (or s_B),
- the new values of variables in $\mathit{frame}(A) \backslash \mathit{frame}(B)$ are as in s_A, and
- the new values of variables in $\mathit{frame}(B) \backslash \mathit{frame}(A)$ are as in s_B.

Dunne defines $A||B$ by

$$\mathit{trm}(A||B) = \mathit{trm}(A) \wedge \mathit{trm}(B)$$
$$\mathit{prd}(A||B) = \mathit{prd}(A) \wedge \mathit{prd}(B)$$

but the latter formula contains an implicit reference to the frames of the commands. It is interesting to note that if A is infeasible, and B is feasible but does not terminate, then $A||B$ is feasible but does not terminate.

We consider first a version of this command for which the frame is the entire set of variables, defined by `pcomp_def` and `pcomprs_def` ; for these, we prove the formulae just mentioned, as `pcomp_prd` and `pcomp_trm`. We also prove as, `pcomp_wlp`, a result (communicated by Dunne)

$$\mathit{wlp}\ (A||B)\ Q\ s = \exists Q_1\ Q_2.(\forall t.Q_1\ t \wedge Q_2\ t \Rightarrow Q\ t) \wedge \mathit{wlp}(A, Q_1)\ s \wedge \mathit{wlp}(B, Q_2)\ s$$

Unusually, we have explicitly referred to states s and t in this statement of the result to make it clear that the choice of Q_1 and Q_2 depends on the state s.

The following definition of $A||B$ takes into account the frames of A and B. Firstly, `pccomb` combines two states (resulting from A and B) if they are compatible.

```
"pccomb ?frA ?frB ?initst (?stA, ?stB) =
   (let compat = ALL str:?frA Int ?frB. ?stA str = ?stB str;
    combst = %str.
       if str : ?frA then ?stA str
       else if str : ?frB then ?stB str else ?initst str
    in if compat then {Term combst} else {})"

"pcompfr ?frA ?A ?frB ?B ?state ==
   let tsA = {st. Term st : ?A ?state};
     tsB = {st. Term st : ?B ?state};
     nont = {NonTerm} Int (?A ?state Un ?B ?state)
   in nont Un UNION (tsA <*> tsB) (pccomb ?frA ?frB ?state)"
```

Here `(tsA <*> tsB)` means the set product of `tsA` and `tsB`. The result `pcomp_chk` is a sanity check that, where the frames of A and B are the set of all strings, this definition is equivalent to the one mentioned in the previous paragraph (a useful check, since our first attempt at the definition above was erroneous). Noting that Dunne's formula $\mathit{prd}(A||B) = \mathit{prd}(A) \wedge \mathit{prd}(B)$ implicitly refers to the frames of the commands, we prove it as `pcompfr_prd`, as follows:

```
pcompfr_prd = "prds (?fA Un ?fB) ?dashed (pcompfr ?fA ?Am ?fB ?Bm) =
   (prds ?fA ?dashed ?Am && prds ?fB ?dashed ?Bm)"
```

5 Conclusion

We have provided an operational model for Dunne's abstract commands and their operators, except that our model does not provide any information about the frame of a command. Based upon this model, we have been able to prove, using the automated prover Isabelle, Dunne's definitions of the abstract command operators, except their frames. That is, we have shown that they follow from our operational model.

We have discussed the problems in including the frame of a command in this work. Briefly, while the frame of a command might be thought of as the set of variables which "might" be set by the command, commands such as $x := x$ (whose frame is $\{x\}$) prevent us from defining the command's frame from its behaviour. We could have attempted to show that the frame of a command (as defined by Dunne) conforms to a rule that the frame contains any variable which can be changed by the command, but this generally seems obvious.

Formalising the various definitions for use in the mechanised prover has highlighted aspects of the specification of commands which need to be considered, but are easily overlooked until one formalises them. Examples of this appear in our discussions about "syntactic" and "semantic" expressions and commands, and about the language in which "syntactic" expressions may be expressed.

Acknowledgement. We wish to thank Steve Dunne for his very great assistance in some lengthy discussions on the topic.

References

1. Steve Dunne, Abstract Commands: A Uniform Notation for Specifications and Implementations, In Computing: The Australasian Theory Symposium (2001), Electronic Notes in Theoretical Computer Science 42. `http://www.elsevier.nl/gej-ng/31/29/23/68/22/show/Products/notes/index.htt#008`
2. Steve Dunne, A Case for General Correctness, submitted.
3. Michael J. C. Gordon. Mechanizing Programming Logics in Higher Order Logic. In G. Birtwistle and P. A. Subrahmanyam (editors), Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989.
4. John Harrison: Formalizing Dijkstra. In Jim Grundy, Malcolm C. Newey (Eds.): Theorem Proving in Higher Order Logics, (TPHOLS'98), Lecture Notes in Computer Science, Vol. 1479, Springer, 1998. 171-188.
5. Dean Jacobs and David Gries. General Correctness: A Unification of Partial and Total Correctness. Acta Informatica 22 (1985), 67-83.
6. Sheng Liang, Paul Hudak, and Mark P Jones. Monad Transformers and Modular Interpreters. In Symposium on Principles of Programming Languages (POPL'95), 1995, 333 343.
7. Greg Nelson. A generalization of Dijkstra's calculus. ACM Transactions on Programming Languages and Systems, 11 (1989), 517-61. Or see DEC (now Compaq) SRC Research Report 16, `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-016.html`
8. Gordon D. Plotkin. A Power-domain construction. SIAM J. Comput. 5 (1976), 452 487.
9. Philip Wadler. The Essence of Functional Programming. In Symposium on Principles of Programming Languages (POPL'92), 1992, 1 14.

Automatic Constraint Calculation using Lax Logic*

Jeremy E. Dawson[1] and Matt Fairtlough[2]

[1] Department of Computer Science and Automated Reasoning Group
Australian National University Canberra ACT 0200, Australia
jeremy@discus.anu.edu.au
[2] Department of Computer Science, The University of Sheffield
Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK
matt@dcs.shef.ac.uk

Abstract. Earlier papers have described how lax logic can be used to develop verified designs, where the lax logic modality is taken to represent some constraint. We show how to use Isabelle's instantiation of variables to obtain elegant proofs of results which are true subject to constraints, and to derive these constraints. We show how this method can be applied to examples of hardware design (where the constraints relate to timing) and to a numerical function (where the constraint is that the machine word length is sufficient).

The Isabelle files required to run the examples in this paper may be found in
`http://www.dcs.shef.ac.uk/~matt/lax/isabelle/Public`

Keywords: lax logic, machine-checked proof, hardware timing

1 Introduction

In [7] Mendler describes Lax Logic and how its single modality may be used to represent that a condition is true up to satisfaction of some constraint. He gives examples, including the calculation of the factorial function using increment and multiplication, which are themselves accurate only up to a constraint. In the factorial example, he takes the constraints on these to be that the word length of the computer is sufficient. In [4] Fairtlough, Mendler and Cheng have described how to analyse the behaviour of hardware, where each particular predicate on the state is true only at certain times, so the constraint is that the predicate is considered at the "right" time. They note that the proof of the "logical" result (e.g, for an and-gate, that if the inputs are true then the output is true) can be separated from the timing constraint (that the inputs have remained true over a certain time interval), and suggest that the proof algorithm could actually be used to *calculate* the constraint. They go on to describe some examples.

In this paper we flesh out this suggestion and consider some examples, including the "latch" example of [4] and the "factorial" example of [7]. We show how we can perform an Isabelle proof in the usual way, but with the constraint initially being a variable, which gets instantiated during the course of the proof. In this way the constraint is generated by the prover – which also proves it correct, in the sense that it implies the corresponding predicate.

Such calculated constraints are not in their simplest form. In Section 3.1 we describe some conversions written to help simplify the constraint which was calculated for the "latch" example, which we expect would be useful for other automatically generated timing constraints.

1.1 Lax Logic

In [3] Fairtlough and Mendler develop "Lax Logic"[1]. This is described as an intuitionistic modal logic, with a single modality $\bigcirc$, which obeys the following axioms:

$$\begin{aligned} \bigcirc R &: M \supset \bigcirc M \\ \bigcirc M &: \bigcirc\bigcirc M \supset \bigcirc M \\ \bigcirc F &: (M \supset N) \supset (\bigcirc M \supset \bigcirc N) \end{aligned}$$

* this work was supported by EPSRC under grant GR/L86180
[1] The logic has been independently invented by Curry [2] and Benton, Bierman & de Paiva [1]

One interpretation of the modality $\bigcirc$ is that it denotes an anonymous constraint or qualification of some sort, that is, $\bigcirc M$ means that M holds under some (unspecified) constraint or qualification. For example, the constraint may be that a certain condition C holds, so $\bigcirc M$ is $C \supset M$, but C is not made explicit.

The latter two axioms may be replaced by the single axiom $\bigcirc E$, and $\bigcirc S$ and $\bigcirc C$ are also useful.

$$\begin{aligned}\bigcirc E &: (M \supset \bigcirc N) \supset (\bigcirc M \supset \bigcirc N)\\ \bigcirc S &: (\bigcirc M \wedge \bigcirc N) \supset \bigcirc(M \wedge N)\\ \bigcirc C &: (M \supset \bigcirc N) \supset (L \supset \bigcirc M) \supset (L \supset \bigcirc N)\end{aligned}$$

Lax Logic can be expressed in a Gentzen-style calculus, with the usual singletons-on-the-right restriction from the Gentzen calculi for Intuitionistic Logic, with the following rules for $\bigcirc$:

$$\frac{\Gamma \vdash M}{\Gamma \vdash \bigcirc M}(\vdash \bigcirc) \qquad \frac{\Gamma, M \vdash \bigcirc N}{\Gamma, \bigcirc M \vdash \bigcirc N}(\bigcirc \vdash)$$

The $\bigcirc$ modality is unusual as it has some properties which are typically $\Box$-like and others that are typically $\Diamond$-like. The usual explanation of this, that $\Box A \longleftrightarrow \Diamond A$ when the underlying Kripke relation R is functional, does not hold in Lax Logic (where $A \longleftrightarrow B$ stands for $(A \supset B) \wedge (B \supset A)$).

Lax Logic also has a Natural Deduction formulation, which we shall use in the rest of this paper. Its rules for $\bigcirc$ appear in Table 2. This formulation is equivalent under the Curry-Howard correspondence to the typing rules for simply-typed λ-calculus with a strong monad [9] an extension of Moggi's computational λ-calculus. The constraint computations carried out in the examples of this paper use the computational rules for the set monad, which are of course consistent with the general equations for a strong monad. We stress this point because our method relies on the extraction of constraint information from an abstract proof; this extraction process involves the abstract computational rules of the calculus, the specific rules of the set monad and the higher-order logic rules for equivalence between propositions. From this viewpoint it is no surprise that the way in which abstract formulas are proved determines the constraints that are extracted from their proofs.

2 Abstract and concrete formulae

In [4] Fairtlough, Mendler and Cheng use Lax Logic to separate aspects of reasoning about hardware circuitry. At the concrete level, aspects such as timing must be taken into account, whereas at the abstract level, only a simplified "boolean" description of the behaviour of devices suffices. For example, at the abstract level, we have "if inputs P and Q are true (high) then output R is also true". Correspondingly, at the concrete level, we have "if inputs P and Q are both high at time t, and remain high until time $t+\delta_1$, then output R goes high no later than $t+\delta_2$, and remains high so long as both P and Q remain high".

Henceforth, following [4], we work in higher-order logic (HOL), and we will define a function $\bigcirc_\forall$ which satisfies the Lax Logic rules for $\bigcirc$. This function is in fact the set monad. Other monads undoubtedly also have their uses within our framework, but we have not yet explored them in any detail. The literature on combining monads, see for example [6], provides a rich source of ideas for extending our method.

Following HOL convention, we will write implication as $\rightarrow$, and following the usage of the Isabelle theorem prover, we will use $P \Longrightarrow Q$ to denote the meta-proposition "Q can be deduced from P" (for which we also use the conventional horizontal bar). Also following Isabelle, we use "!!" as a universal quantifier understood at the meta-level.

2.1 Translating logic into the concrete/abstract format

Consider as an example a concrete formula such as $P\ s$, which means that signal P is high (true) at time s, and likewise a second formula $Q\ t$. The corresponding abstract formulae are just P and Q, whose conjunction is $P \wedge Q$. The concrete formula expressing the conjuction is $P \sqcap Q$, defined by $(P \sqcap Q)(s,t) \equiv P\ s \wedge Q\ t$.

More typically we would want to express a formula containing an arbitrary constraint, for example $\forall s. s \geq 5 \rightarrow P\ s$. Here one could separate the concrete and abstract by writing $\{s \mid s \geq 5\} \subseteq \{s \mid P\ s\}$. We define the concrete modality $\bigcirc_\forall$ by $\bigcirc_\forall P c \equiv \forall s. c s \rightarrow P\ s$; then $\bigcirc_\forall$ obeys the Lax Logic axioms for $\bigcirc$ given earlier. We can now write this last constraint as $\bigcirc_\forall P(\lambda s. s \geq 5)$.

In both these expressions, i.e. $(P \sqcap Q)(s,t)$ and $\bigcirc_{\forall} P(\lambda s.s \geq 5)$, the concrete part is the (last curried) argument, and the abstract part is the rest.

Note, however, that in the latter example, the abstract part includes the Lax Logic modality $\bigcirc_{\forall}$.

2.2 The translation used in the Isabelle/HOL implementation

As an alternative to the above, we tried replacing predicates by the corresponding sets (where the predicate is the characteristic function of the set), replacing $P\ s$ by $s \in P$, and so forth. After experimenting with both styles in Isabelle/HOL, we proceeded with the set-based implementation, since it renders the derivations of the rules corresponding to $\bigcirc R$, $\bigcirc M$, $\bigcirc F$, $\bigcirc E$ and $\bigcirc S$ particularly transparent (these are given later, in Table 2).

In the set-based notation, the definition of $\sqcap$ becomes $(s,t) \in (P \sqcap Q) \equiv s \in P \wedge t \in Q$ and the formula $\forall s.s \geq 5 \to s \in P$ is rewritten as $\{s \mid s \geq 5\} \in \bigcirc_{\forall} P$, where $c \in \bigcirc_{\forall} P \equiv \forall s.s \in c \to s \in P$. This last definition may be more simply written $c \in \bigcirc_{\forall} P \equiv c \subseteq P$.

Thus, in this formulation, the concrete formula is of the form $c \in P$, where the abstract formula is simply P. Henceforth in this paper, we will follow our Isabelle implementation and write formulae using the set-based formulation rather the function-based one.

2.3 Calculating constraints

Intuitively, the formula $s \in P$ may be seen in the light that s is a concrete witness of the abstract formula P: it gives an instance where P holds. Likewise, in the formula $(s,t) \in (P \sqcap Q)$, (s,t) gives an instance where $P \sqcap Q$ holds.

In reasoning about timing of logical circuits, when P and Q are "primitive" (eg gate inputs or outputs), s and t will be of type "time". Note, however, that the witnesses for "compound" (abstract) formulae, such as $P \sqcap Q$, will be of a different type. For example, (s,t) is of type "pair of times".

As noted above, the abstract formula $P \wedge Q$ corresponds to the concrete formula $(s,t) \in (P \sqcap Q)$. Likewise, the following two (Natural Deduction style) rules of inference (of which the second is expressed in two forms) correspond.

$$\frac{P \quad Q}{P \wedge Q} \qquad\qquad \frac{s \in P \quad t \in Q}{(s,t) \in (P \sqcap Q)} \qquad\qquad \frac{\textbf{fst}\ z \in P \quad \textbf{snd}\ z \in Q}{z \in (P \sqcap Q)}$$

Note here that the rule calculates the witness for the conclusion from the witnesses for the premises. Alternatively, if the rule is used in a backward proof, the second form calculates the required witnesses for the two subgoals in terms of the required witness for the conclusion.

Similarly, we can find a concrete equivalent for all the rules of Lax Logic. Table 1 gives many examples for the intuitionistic logic rules. In some cases the rules are in the specific form used in the Isabelle/HOL system. The third column gives the name of the rule (or of a close equivalent) in the implementation.

Note that, because a concrete formula specifies a witness, a simpler rule than `Sumd_Ep` is available for eliminating an abstraction $\bigsqcup y.$, namely the inverse of the `Sumd_Ip` rule.

For the rules using the $\bigcirc_{\forall}$ operator, a selection of the rules is shown in Table 2. Note that the rules labelled $(\bigcirc_{\forall} R)$, $(\bigcirc_{\forall} M)$, etc, are equivalent to the axioms $(\bigcirc R)$, $(\bigcirc M)$, etc listed earlier, but they are in the form of Natural Deduction inference rules. Note that the binary infix operator ‘ (written op ‘ as a curried 2-argument function) is given by op ‘ $f S \equiv f`S \equiv \{f x \mid x \in S\}$.

2.4 The "Tiny" example

In [4] the following simple example is given. The concrete formulae (expressed in terms of sets rather than predicates) are

$\psi_1 : \forall s.s \geq 5 \to s \in P_1 a$
$\psi_2 : \forall sy.s \geq 9y \to s \in P_2(fy)$
$\psi_3 : \forall ts y_1 y_2.(t \geq s + 35 \wedge s \in P_1 y_1 \wedge s \in P_2 y_2) \to t \in Q(g(y_1, y_2))$

Table 1. Some logical rules with concrete equivalents (Natural Deduction format)

Abstract	Concrete	HOL rule
$\dfrac{\begin{array}{l}P \wedge Q\\ P \Longrightarrow Q \Longrightarrow R\end{array}}{R}(\wedge E)$	$\dfrac{\begin{array}{l}x \in P \sqcap Q\\ \textbf{fst}\ x \in P \Longrightarrow \textbf{snd}\ x \in Q \Longrightarrow R\end{array}}{R}(\sqcap E)$	**andd_Es**
$\dfrac{P \quad Q}{P \wedge Q}(\wedge I)$	$\dfrac{s \in P \quad t \in Q}{(s,t) \in (P \sqcap Q)}(\sqcap I)$	**andd_I**
$\dfrac{P \Longrightarrow Q}{P \to Q}(\to I)$	$\dfrac{!!x.x \in P \Longrightarrow fx \in Q}{f \in P \sqsupset Q}(\sqsupset I)$	**impd_Ip'**
$\dfrac{P \to Q \quad P}{Q}(\to E)$	$\dfrac{f \in P \sqsupset Q \quad p \in P}{fp \in Q}(\sqsupset E)$	**impd_Ep'**
$\dfrac{P \to Q \quad Q \to R}{P \to R}$	$\dfrac{f \in P \sqsupset Q \quad g \in Q \sqsupset R}{g \circ f \in P \sqsupset R}$	**impd_trans**
$\dfrac{\forall y.Py}{Px}(\forall E)$	$\dfrac{f \in \prod y.Py}{fx \in Px}$	**Prodd_Ep**
$\dfrac{!!x.Px}{\forall y.Py}(\forall I)$	$\dfrac{!!x.fx \in Px}{f \in \prod y.Py}$	**Prodd_Ip**
$\dfrac{Px}{\exists y.Py}(\exists I)$	$\dfrac{z \in Pw}{(w,z) \in \bigsqcup y.Py}$	**Sumd_Ip**
$\dfrac{\exists y.Py \quad !!x.Px \Longrightarrow R}{R}(\exists E)$	$\dfrac{\begin{array}{l}(w,z) \in \bigsqcup y.Py\\ !!u\,x.u \in Px \Longrightarrow fxu \in R\end{array}}{fwz \in R}$	**Sumd_Ep**

Firstly, these are rewritten to use $\bigcirc_\forall$:

$\psi_1^\sharp : \{s \geq 5\} \in \bigcirc_\forall(P_1 a)$
$\psi_2^\sharp : \forall y.\{s \geq 9y\} \in \bigcirc_\forall(P_2(fy))$
$\psi_3^\sharp : \forall y_1 y_2 s_1 s_2 . s_1 \in P_1 y_1 \wedge s_2 \in P_2 y_2 \to \{t \mid s_1 = s_2 \wedge t \geq s_1 + 35\} \in \bigcirc_\forall(Q(g(y_1, y_2)))$

From the above we can delete the concrete information (i.e, that before $\in$, and the quantified variables s_1 and s_2) to get the following formulae which are abstract, but expressed in terms of Lax Logic.

$\psi_1^{\sharp 2} : \bigcirc_\forall(P_1 a)$
$\psi_2^{\sharp 2} : \forall y.\bigcirc_\forall(P_2(fy))$
$\psi_3^{\sharp 2} : \forall y_1 y_2 . P_1 y_1 \wedge P_2 y_2 \to \bigcirc_\forall(Q(g(y_1, y_2)))$

Where it is required to prove $\exists v.\bigcirc_\forall(Qv)$, the abstract Lax Logic proof would run as follows.

$$\dfrac{\dfrac{\dfrac{\psi_3}{\forall y_1 y_2 . P_1 y_1 \wedge P_2 y_2 \to \bigcirc_\forall(Q(g(y_1, y_2)))}}{P_1 a \wedge P_2(fb) \to \bigcirc_\forall(Q(g(a, fb)))}(\forall E) \quad \dfrac{\dfrac{\psi_1}{\bigcirc_\forall(P_1 a)} \quad \dfrac{\dfrac{\psi_2}{\forall y.\bigcirc_\forall(P_2(fy))}}{\bigcirc_\forall(P_2(fb))}(\forall E)}{\bigcirc_\forall(P_1 a \wedge P_2(fb))}(\bigcirc_\forall S)}{\dfrac{\bigcirc_\forall(Q(g(a, fb)))}{\exists v.\bigcirc_\forall(Qv)}(\exists I)}(\bigcirc_\forall E)$$

Note that we use $(\bigcirc_\forall S)$ and $(\bigcirc_\forall E)$ for rules which are equivalent to the corresponding axioms shown in §1.1.

To do the concrete constraint calculation, we would rewrite the formulae again using the logic of the implementation such that the abstract formulae appear after the $\in$, as follows (we also show an intermediate step in the rewriting for ψ_3):

Table 2. Some modal rules with concrete equivalents

Abstract	Concrete	HOL rule
$\dfrac{P}{\bigcirc_{\forall}P}(\bigcirc_{\forall}R)$	$\dfrac{x \in P}{\{x\} \in \bigcirc_{\forall}P}$	`Oall_unit'`
$\dfrac{P \to Q}{\bigcirc_{\forall}P \to \bigcirc_{\forall}Q}(\bigcirc_{\forall}F)$	$\dfrac{f \in P \sqsupset Q}{\texttt{op}\ \text{`}\ f \in \bigcirc_{\forall}P \sqsupset \bigcirc_{\forall}Q}$	`Oall_F_image`
$\dfrac{\bigcirc_{\forall}(\bigcirc_{\forall}P)}{\bigcirc_{\forall}P}(\bigcirc_{\forall}M)$	$\dfrac{S \in \bigcirc_{\forall}(\bigcirc_{\forall}P)}{\bigcup S \in \bigcirc_{\forall}P}$	`Oall_M_Union`
$\dfrac{\bigcirc_{\forall}P \quad \bigcirc_{\forall}Q}{\bigcirc_{\forall}(P \wedge Q)}(\bigcirc_{\forall}S)$	$\dfrac{S \in \bigcirc_{\forall}P \quad T \in \bigcirc_{\forall}Q}{S \times T \in \bigcirc_{\forall}(P \sqcap Q)}$	`Oall_S_times`
$\dfrac{P \to \bigcirc_{\forall}Q \quad \bigcirc_{\forall}P}{\bigcirc_{\forall}Q}(\bigcirc_{\forall}E)$	$\dfrac{f \in P \sqsupset \bigcirc_{\forall}Q \quad S \in \bigcirc_{\forall}P}{\bigcup(f\text{`}S) \in \bigcirc_{\forall}Q}$	`Oall_ext''`

$\psi_1^\sharp : \{s \geq 5\} \in \bigcirc_{\forall}(P_1 a)$
$\psi_2^\sharp : \lambda y.\{s \geq 9y\} \in \prod y.\bigcirc_{\forall}(P_2(fy))$
$\psi_3^\sharp$ (intermediate form) :
$\quad \forall y_1 y_2.\lambda(s_1, s_2).\{t \mid s_1 = s_2 \wedge t \geq s_1 + 35\} \in P_1 y_1 \wedge P_2 y_2 \to \bigcirc_{\forall}(Q(g(y_1, y_2)))$
$\psi_3^\sharp : \lambda y_1 y_2.\lambda(s_1, s_2).\{t \mid s_1 = s_2 \wedge t \geq s_1 + 35\} \in \prod y_1 y_2.P_1 y_1 \wedge P_2 y_2 \to \bigcirc_{\forall}(Q(g(y_1, y_2)))$

We would then repeat the proof, using rules on the right-hand sides of Tables 1 and 2. In the implementation in Isabelle/HOL, we use the rules listed in place of the logic rules. The file `Tiny2.ML` gives a proof of the goal $p : \bigsqcup v.\bigcirc_{\forall}(Qv)$, for some appropriate p, which precisely mimics the abstract proof above, using the corresponding rules shown in the Tables. In Isabelle we can leave p unspecified – it is a variable in the goal – and the proof process in Isabelle calculates it, by instantiating the variable.

2.5 The "Latch" example

This example deals with a latch, with two inputs r_{in} and s_{in}, and two outputs q_{out} and $\overline{q_{out}}$. The latch is constructed using two cross-coupled NOR-gates, and has the "memory" property that,

- if one input is low and remains so, and
- the other input is high and goes low,
- then (so long as both inputs remain low) the outputs remain the same as they were when one input was high.

This is possible because of the feedback from each gate to the other in the design of the latch circuit, and because the gates have a finite delay.

The rules governing the latch, as relevant for this proof, are θ_1, θ_2 and θ_3, and the initial conditions are θ_{p1} and θ_{p2}. The notation $(s_a, t_a) \in (\!| r_{in} |\!)$ means that, for all times from s_a to t_a inclusive, the signal r_{in} is high. It may be noted (from the description above of the latch functionality) that the outputs should remain steady regardless of whether or when r_{in} goes low (in fact the outputs q_{out} and $\overline{q_{out}}$ should remain low and high respectively).

$\theta_{p1} : (s_a, t_a) \in (\!| r_{in} |\!)$
$\theta_{p2} : \forall t \geq s_a.(s_a, t) \in (\!| \rceil s_{in} |\!)$ [2]
$\theta_1 : \forall s\, t.(s, t) \in (\!| r_{in} |\!) \to (s + d_1, t + D_1) \in (\!| \rceil q_{out} |\!)$
$\theta_2 : \forall s_1\, t_1\, s_2\, t_2.(s_1, t_1) \in (\!| \rceil s_{in} |\!) \wedge (s_2, t_2) \in (\!| \rceil q_{out} |\!) \to$
$\quad (\max s_1\, s_2 + d_2, \min t_1\, t_2 + D_2) \in (\!| \overline{q_{out}} |\!)$

[2] $\rceil$ is used for not, as in [4].

$\theta_3 : \forall s\, t.(s, t) \in \llparenthesis \overline{q_{out}} \rrparenthesis \to (s + d_1, t + D_1) \in \llparenthesis \rceil q_{out} \rrparenthesis$

We translate these into the form which separates the abstract and concrete to get

$\theta_{p1} : (s_a, t_a) \in \llparenthesis r_{in} \rrparenthesis$
$\theta_{p2} : \{(s_a, t) \mid t \geq s_a\} \in \bigcirc_\forall \llparenthesis \rceil s_{in} \rrparenthesis$
$\theta_1 : \lambda(s, t).(s + d_1, t + D_1) \in \llparenthesis r_{in} \rrparenthesis \sqsupset \llparenthesis \rceil q_{out} \rrparenthesis$
$\theta_2 : \lambda((s_1, t_1), (s_2, t_2)).(\max s_1\, s_2 + d_2, \min t_1\, t_2 + D_2) \in \llparenthesis \rceil s_{in} \rrparenthesis \sqcap \llparenthesis \rceil q_{out} \rrparenthesis \sqsupset \llparenthesis \overline{q_{out}} \rrparenthesis$
$\theta_3 : \lambda(s, t).(s + d_1, t + D_1) \in \llparenthesis \overline{q_{out}} \rrparenthesis \sqsupset \llparenthesis \rceil q_{out} \rrparenthesis$

The reasoning goes like this:

(a) Because r_{in} is high (by θ_{p1}), q_{out} is low (by θ_1).
(b) Because s_{in} is low (by θ_{p2}) and q_{out} is low (by step (a) or (c)), $\overline{q_{out}}$ is high (by θ_2).
(c) Because $\overline{q_{out}}$ is high (by step (b)), q_{out} is low (by θ_3).
(d) Now go back to step (b)

Because of the inertiality and delay of gates, this cyclic argument corresponds to the physical process that keeps q_{out} low permanently. Note that it requires that s_{in} be low permanently, which is expressed by the concrete version of θ_{p2}.

The proof can be broken into three parts: step (a), steps (b) – (c) – (d), and the integration of these two parts. We first discuss steps (b) – (c) – (d).

The abstract result we prove is $\llparenthesis \rceil q_{out} \rrparenthesis \Longrightarrow \bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis$. This result is trivial – for example, use the axiom $\bigcirc_\forall R$ – but we need a proof which expresses the reasoning above. This is because the reasoning above gives (loosely) "if q_{out} is low on a certain interval, then it is low on another (slightly later) interval". Then, ultimately, q_{out} is low permanently.

We call that result `latch_step`. Here is the proof of the abstract result using the abstract connectives, following the proof in [4, §3, equations (2) to (5)].

$$\dfrac{\dfrac{\theta_3}{\llparenthesis \overline{q_{out}} \rrparenthesis \to \llparenthesis \rceil q_{out} \rrparenthesis} \quad \dfrac{\dfrac{\theta_2}{\llparenthesis \rceil s_{in} \rrparenthesis \land \llparenthesis \rceil q_{out} \rrparenthesis \to \llparenthesis \overline{q_{out}} \rrparenthesis} \quad \dfrac{\dfrac{\theta_{p2}}{\bigcirc_\forall \llparenthesis \rceil s_{in} \rrparenthesis} \quad \dfrac{\dfrac{\text{Assume}}{\llparenthesis \rceil q_{out} \rrparenthesis}}{\bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis}\,(\bigcirc_\forall R)}{\bigcirc_\forall (\llparenthesis \rceil s_{in} \rrparenthesis \land \llparenthesis \rceil q_{out} \rrparenthesis)}\,(\bigcirc_\forall S)}{\bigcirc_\forall \llparenthesis \overline{q_{out}} \rrparenthesis}\,(\bigcirc_\forall F)}{\bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis}\,(\bigcirc_\forall F)$$

This tree proves $\llparenthesis \rceil q_{out} \rrparenthesis \Longrightarrow \bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis$, and $\llparenthesis \rceil q_{out} \rrparenthesis \to \bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis$ follows trivially.

As in the "Tiny" example, this proof can be translated to a proof of the concrete goal $p \in \llparenthesis \rceil q_{out} \rrparenthesis \sqsupset \bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis$ for some appropriate p, which is calculated in doing the Isabelle proof.

The file `Memory2.ML` gives a proof which precisely mimics the abstract proof above, using the corresponding rules shown in Tables 1 and 2.

A rule is now needed to incorporate the repeated use of `latch_step` (which incorporates steps (b) to (d) in the proof outline above). The *inductive* property of a set is defined in [4] thus:

$$\text{Ind } P \equiv x \in P \sqsupset (P \sqsupset \bigcirc_\forall P) \sqsupset \bigcirc_\forall P \tag{1}$$

(where x is a specific value, given in [4]), and it is proved that Ind $\llparenthesis P \rrparenthesis$ holds for any P. It seems difficult to describe the proof of this result as an abstract proof where the implementation calculates the constraint automatically. The proof seems tailored towards the desired constraint much more than in the "Tiny" example, or in the proof of `latch_step`. Proofs of this result are given in `Induction.ML`. Later, in §2.6, we describe a different way of using `latch_step` to prove the required result, first focussing only on the proof at the abstract level.

We can now use the induction rule $\llparenthesis P \rrparenthesis \to (\llparenthesis P \rrparenthesis \to \bigcirc_\forall \llparenthesis P \rrparenthesis) \to \bigcirc_\forall \llparenthesis P \rrparenthesis$ in the following proof.

$$\dfrac{\dfrac{\dfrac{\theta_1}{\llparenthesis r_{in} \rrparenthesis \to \llparenthesis \rceil q_{out} \rrparenthesis} \quad \dfrac{\theta_{p1}}{\llparenthesis r_{in} \rrparenthesis}}{\llparenthesis \rceil q_{out} \rrparenthesis}(\to E) \qquad \dfrac{(latch\_step)}{\llparenthesis \rceil q_{out} \rrparenthesis \to \bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis}}{\bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis}(induction\ rule)$$

Again, we convert this proof using Tables 1 and 2, and the constraint is calculated for us; see `Memory2.ML`.

2.6 Alternative Approach

In part of the proof described above, we considered an abstract proof and performed the corresponding concrete proof in Isabelle, letting Isabelle generate the relevant constraint. However the proof of the induction rule did not follow this pattern; rather, the proof seemed to be targeted at the desired constraint. We tried to improve this by proving forwards from the rule `latch_step`.

Firstly, two intervals which overlap can be combined to form a single interval. We can express this as $x \in \llparenthesis P \rrparenthesis \Rightarrow y \in \llparenthesis P \rrparenthesis \Rightarrow S \subseteq \llparenthesis P \rrparenthesis$ where $S = \emptyset$ if x and y do not overlap, and $S = \{x \cup y\}$ if they do. The abstract version of the rule, called `During_overlap'`, is $\llparenthesis P \rrparenthesis \to \llparenthesis P \rrparenthesis \to \bigcirc_\forall \llparenthesis P \rrparenthesis$.

The result `latch_step` expresses that from an interval in $\llparenthesis \rceil q_{out} \rrparenthesis$ we derive a set of intervals in $\llparenthesis \rceil q_{out} \rrparenthesis$. Of this set of intervals, those that overlap the initial interval can be joined with it, giving a set of intervals which are at least as long as the initial interval.

Here is the abstract version of this. Note that the assumption appears twice, which reflects the fact that the initial interval is used twice, once to generate a set of intervals, and once to join with each of them. The result obtained, also $\llparenthesis \rceil q_{out} \rrparenthesis \to \bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis$, is called `During_extend`.

$$\dfrac{\dfrac{\dfrac{\text{Assume}}{\llparenthesis \rceil q_{out} \rrparenthesis}}{\llparenthesis \rceil q_{out} \rrparenthesis \to \bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis}\texttt{During\_overlap'} \qquad \dfrac{\dfrac{\text{Assume}}{\llparenthesis \rceil q_{out} \rrparenthesis}}{\bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis}\texttt{latch\_step}}{\bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis}(\bigcirc_\forall E)$$

This tree proves $\llparenthesis \rceil q_{out} \rrparenthesis \Longrightarrow \bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis$, and $\llparenthesis \rceil q_{out} \rrparenthesis \to \bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis$ follows trivially.

At this point we realized that we want our subset S of $\llparenthesis \rceil q_{out} \rrparenthesis$ to be closed under "shrinking from the right"; that is, if $(a, b) \in S$ and $b' \leq b$ then $(a, b') \in S$. It is obvious that if $(a, b) \in \llparenthesis P \rrparenthesis$ and $b' \leq b$ then $(a, b') \in \llparenthesis P \rrparenthesis$, that is, if $(a, b) \in \llparenthesis P \rrparenthesis$, then $\{(a, b') \mid b' \leq b\} \in \bigcirc_\forall \llparenthesis P \rrparenthesis$, so this gives the result `During_shrink_right`, which is, abstractly, $\llparenthesis P \rrparenthesis \to \bigcirc_\forall \llparenthesis P \rrparenthesis$. Using $(\bigcirc_\forall C)$ we combine `During_shrink_right` and `During_extend` to give `During_extend_all`.

By `During_extend_all` we have $(a, b) \in \llparenthesis \rceil q_{out} \rrparenthesis \Rightarrow S \subseteq \llparenthesis \rceil q_{out} \rrparenthesis$ where S contains intervals which extend (a, b) to the right. We want to repeat this *ad infinitum*, forever accumulating larger intervals in S. To do this we used Isabelle's inductive definition facility, as follows. The code shown defines `rep h x` as the smallest set satisfying rules `repI1` and `repI2`, and it is easily proved, as `rep_fp`, that $rep \in (P \sqsupset \bigcirc_\forall P) \sqsupset (P \sqsupset \bigcirc_\forall P)$.

```
consts rep :: "('a => 'a set) => ('a => 'a set)"
inductive "rep h x"
  intrs
    repI1 "x : rep h x"
    repI2 "[| z : h y ; y : rep h x |] ==> z : rep h x"
```

Finally, we combine `rep_fp` with `During_extend_all` to get `latch_abs_rep`. Note that, as abstract results, `latch_step`, `During_extend`, `During_extend_all` and `latch_abs_rep` are all the same, ie $\llparenthesis \rceil q_{out} \rrparenthesis \to \bigcirc_\forall \llparenthesis \rceil q_{out} \rrparenthesis$. However they have different (and successively more complex) constraints.

Having obtained this result using (we thought) the correct logic, we then set out to check that the constraint obtained was equivalent to the one found previously – in fact it turned out that `latch_abs_rep` was a stronger result (had a weaker constraint) than the earlier one.

2.7 Comment on Proof Style

The file `Memory.ML` gives proofs of the "latch_step" and the final result. The proofs of corresponding results in `Memory2.ML` and `Memory3.ML` are very much shorter. While the results proved in `Memory2.ML` and `Memory3.ML` contain somewhat different constraints from those proved in `Memory.ML`, it is not unduly difficult to show the equivalence between them (see §3).

In fact we can attribute our success in streamlining the proofs to the concept outlined in [4, p. 4], where the authors say

> *Our contribution.* Our approach involves maintaining a close connection between abstraction (the deductive dimension) and constraints (the algorithmic dimension). The algorithmic aspect corresponds to the *calculation* of constraints. ...

In the proofs in `Memory2.ML` and `Memory3.ML`, we have first performed a proof by looking only at the abstract parts of the terms. In fact we did this by working out the corresponding proof in Lax Logic as shown in the proof trees above, and then translated this using Tables 1 and 2. While we were concentrating solely on the abstract "side" of the formulae (on the right-hand sides of the '$\in$' in the rules in the second column of the tables), Isabelle was constructing (or calculating) the constraints on the left-hand sides of the '$\in$'.

As can be seen in `Memory2.ML` and `Memory3.ML`, this made for quite short proofs. Only then did we turn to the constraints, proving that they were equivalent to the desired constraints (that is, the constraints found in the results in `Memory.ML`). This whole process made for shorter proofs than those found in `Memory.ML`.

2.8 Different proofs

It was observed earlier that the `latch_step` result could be proved (abstractly) simply by applying $\bigcirc_\forall R$. This would give, however, a different concrete result (i.e, with different timing constraint term), which would not have been what we wanted. For another example, the important result `rep_fp` corresponds to the trivial abstract result $(P \to \bigcirc_\forall P) \to (P \to \bigcirc_\forall P)$, but the trivial proof would give a different concrete result. This illustrates that taking an automated proof of an abstract result and then converting it to a proof of the concrete result will often not give the desired constraint. Likewise, we have noted that several distinct concrete results, with different proofs, correspond to the same abstract result as does `latch_step`. It is necessary to take into account not only whether the abstract result is provable but *how* it may be proven. In this sense our abstract rules are intended to be used more like tactics than theorems.

2.9 The "Factorial" example

In [7], Mendler gives some practical examples. We consider his "Factorial" example, where he defines an implementation of the factorial in terms of implementations of an "increment" function and of multiplication, both of which can handle integers only up to a certain size.

In this example, the factorial function is implemented using the usual recursive definition, but using functions i (for incrementing) and m (for multiplication), which implement the mathematical successor and multiplication functions subject to constraints.

The results are in the file `Factorial.ML`, with definitions in the file `Factorial.thy`. The functions Mendler used are as follows:

`fact`, `Suc` and `*` are the true factorial, successor and multiplication functions,

`i` and `m` are implementations of the successor (or increment) and multiplication functions, which are correct up to constraints

`cnt` is an implementation of the successor function—in effect, $cnt\ n = i^n\ 1$

```
cnt_0 "cnt i m 0 = Suc 0"
cnt_Suc "cnt i m (Suc n) = i (cnt i m n)"
```

`fac` is an implementation of the factorial function, using `cnt`

```
fac_0 "fac i m 0 = 1"
fac_Suc "fac i m (Suc n) = m (cnt i m n) (fac i m n)"
```

Note that cnt and fac are defined in terms of i and m, which is a detail necessary in the source code but omitted from the Figures 4-7 and 4-8 in [7]. We also note that this implementation assumes that a given argument can be correctly identified as being equal to Suc n – there is no constraint for the accuracy of this step.

Figure 4-7 in [7] gives an inductive proof that the function cnt implements the successor function, based on a premise that i also implements the successor function. We followed this proof, generally substituting rules on the right-hand sides of Tables 1 and 2 for those on the left-hand sides. As the proof in Figure 4-7 is by induction on the natural numbers, we show nat_induct, the concrete equivalent of the induction axiom.

$$\frac{P\ 0 \quad \forall n.P\ n \to P(Suc\ n)}{\forall n.P\ n} \qquad \frac{a \in P\ 0 \quad f \in \prod n.P\ n \to P(Suc\ n)}{nat\_rec\ a\ f \in \prod n.P\ n}$$

where *nat_rec* is defined by

```
nat_rec f g 0 = f
nat_rec f g (Suc n) = g n (nat_rec f g n)
```

Recall that previously, where the abstract expression was a boolean condition, the constraint was that the condition would hold only at certain times. Therefore the concrete quantity corresponding to the abstract condition was the set of times at which the condition would hold.

Here the constraints are themselves simply boolean conditions. To use the same concrete rules as previously, we use a function ι of type *bool* $\to$ *unit set*, where $\iota(true) = \{()\}$ and $\iota(false) = \{\}$. (*unit* is the type with just one value, '()', and so *unit set* has two values, $\{()\}$ and $\{\}$).

We first proved a goal of the form

$$\frac{f \in \prod n(\bigcirc_\forall(\iota(i\ n = Suc\ n)))}{?g \in \prod n(\bigcirc_\forall(\iota(cnt\ i\ m\ n = Suc\ n)))}$$

where $?g$ denotes a variable which would (usually) become instantiated during the course of the proof. Observe that $?g$ could always be instantiated by the function $g\ n = \{\}$, but the proof process gives us the "largest" possible g (in terms of f). That is, assuming $f\ n$ is $\{()\}$ whenever $i\ n = Suc\ n$ holds, then $f\ n$ is $\{()\}$ whenever it follows (according to the proof in Figure 4-7)) that $cnt\ i\ m\ n = Suc\ n$. The Isabelle proof instantiates g to $nat\_rec\{()\}(\lambda nx. \bigcup((\lambda y.f(Suc\ n))`x))$ This gave the result deriv2.

From here we look at the proof in Figure 4-8 of [7], and attempt to prove a goal of the form

$$\frac{f \in \prod n.(\bigcirc_\forall(\iota(i\ n = Suc\ n))) \quad g \in \prod n_1 n_2.(\bigcirc_\forall(\iota(m\ n_1\ n_2\ = n_1 * n_2)))}{?h \in \prod n(\bigcirc_\forall(\iota(fac\ i\ m\ n = fact\ n)))}$$

The proof in Figure 4-8 is not quite complete for our purpose, as it shows only a proof of the premises of the main inductive step. But using Figure 4-8 of [7] plus that inductive step we obtain the result deriv4 in Factorial.ML.

We then applied these results, choosing, as constraints for the functions i and m, the conditions that the result fits into a word length of w bits. So our constraint for $i\ n = Suc\ n$ is $Suc\ n < 2^w$, and our constraint for $m\ n_1\ n_2 = n_1 * n_2$ is $n_1 * n_2 < 2^w$. (These constraints are expressed in the functions I and M, defined by I_def and M_def).

We then simplified the constraint of deriv4. This involved a few theorems which may be generally useful in dealing with the *unit set* type, and some theorems specific to the constraint being simplified. Finally we obtained the theorem deriv4'1, which gives the constraint $(n = 0 \mid fact\ n < 2^w)$. Expanding the definitions of the connectives $\bigcirc_\forall$, $\prod$, ι and the definitions of I and M, and applying a little automatic simplification, we obtain a correctness theorem of the form

$$\frac{\forall n.\ Suc\ n < 2^w \to i\ n = Suc\ n \quad \forall n_1, n_2. n_1 * n_2 < 2^w \to m\ n_1\ n_2 = n_1 * n_2}{\forall n.(n = 0 \to fac\ i\ m\ 0 = fact\ 0) \wedge (fact\ n < 2^w \to fac\ i\ m\ n = fact\ n)}$$

In fact we weren't expecting the disjunct $n = 0$ in the constraint of deriv4'1, but it is clear that it should be there. The reason is that *fac i m* 0 can be evaluated without using the implementation of i or m – it is defined as 1. (An alternative definition of *fac* might have been $fac\ i\ m\ 0 = i\ 0$). We mention this only because it is reassuring to get an unexpected result which, on reflection, proves correct!

3 Dealing with the resulting constraints

As mentioned, this approach has the significant advantage that constraints can be calculated automatically. Typically, at each step of the proof, the constraint attached to the current proof becomes larger and larger. It may therefore require significant effort to simplfy the constraint to a point where it is useful. For example, the constraint appearing in `deriv4` (*before* substituting the actual constraints on `i` and `m` (increment and multiplication) for f and g) is

$$nat\_rec\{()\}(\lambda nx.nat\_rec\{()\}(\lambda nx.\bigcup((\lambda y.f(Suc\ n))``x))n \cap g(Suc\ n)(fact\ n) \cap x)$$

In the case of the "Latch" example, part of the constraint was the expression

$$\begin{aligned}&(\forall t_1.t + D_1 \leq t_1 \rightarrow (\exists s_2.s + d_1 \leq s_2 \wedge s_2 \leq t_1 \wedge \\ &\ (\exists t_2.t_1 < t_2 \wedge (\exists a.s_2 = \max a\ (s + d_1) + d_2 + d_1 \wedge \\ &\ (\exists b.t_2 = \min b\ t_1 + D_2 + D_1 \wedge s \leq a \wedge a \leq b)))))\end{aligned}$$

which can be simplified to

$$2 * d_1 + (s + d_2) \leq t + D_1 \wedge (0 < D_2 \mid 0 < D_1)$$

We felt that the form of the constraint produced would be likely to recur in other cases in connection with timing constraints for hardware, and so we developed some conversions to assist the simplification.

This simplification requires removing all five of the quantifiers. We produced four conversions which in various ways remove quantifiers automatically. All five quantifiers in this example were removed by the conversions we produced, which suggests that they may be of some general use.

The actual simplification, with and without use of the conversions, is in the file `Memory2.ML`.

3.1 Conversions

The concept of a "conversion" is found in the HOL theorem prover, see [5, Chapter 13], where a *conversion* is a function of type `term -> thm`, which takes a term t to a theorem $t = t'$ (which can be used to rewrite t to t'). A *conversional* is a function which acts on or modifies conversions. For example, if *conv* t is $t = t'$ (to rewrite t to t'), and *conv'* t' is $t' = t''$ (to rewrite t' to t'') then (*conv* `THENC` *conv'*) t is $t = t''$ (to rewrite t to t''). The conversional `SUB_CONV` applies a conversion to the immediate subterms of a term. These conversionals may be used to program various more complex strategies for rewriting (where possible) subterms of a term, as described in [5, §13.2].

In HOL, a conversion *conv* that "fails", in the sense that *conv* t finds no t' such that $t = t'$, can be programmed either to return $t = t$ or to raise an exception (and there are functions to change one sort of conversion to the other). We implemented this concept slightly differently. Our conversion has type `cterm -> thm option` (a `cterm` is a term "certified" to be type-correct). Given t, if an equivalent but different expression t' is not found, then (normally) `NONE` is returned (we say the conversion "fails") though some conversions will return $t = t$ (we say they "succeed *trivially*")

The structure `Conv` (file `conv.ML`) contains many functions relating to conversions. (All have close counterparts in HOL [5, Chapter 13]). For example, the following three functions apply c_1 and/or c_2, but with different results depending on their success or failure.

c_1 `THENC` c_2 applies c_1 and then c_2, failing if either c_1 or c_2 fails,

c_1 `BOTHC` c_2 behaves just as c_2 if c_1 fails, but if c_1 succeeds it tries c_2 on the result of c_1; it fails only if both c_1 and c_2 fail,

c_1 `ORELSEC` c_2 tries c_1, and tries c_2 if c_1 fails.

For a single conversion c,

`TRY_CONV` c t tries c t, succeeding with $t = t$ (succeeding trivially) if c t fails

`REPEAT1C` c repeats c until it fails, succeeding only if c succeeds at least once

`REPEATC` c tries c repeatedly until it fails, but always itself succeeds

Clearly a conversion which can succeed trivially should not be `REPEATC`-ed, or an infinite loop may result.

For a conversion c, where t is $t_1\ t_2$, `COMB_CONV` $c\ t$ applies c to t_1 and to t_2; where t is $\lambda x.t'$, `ABS_CONV` $c\ t$ applies c to t'.

With these building blocks, we can define `TOPDN_CONV` and `BOTUP_CONV`; `TOPDN_CONV` $c\ t$ applies c to all subterms of t, without repetition, in top-down order, and likewise `BOTUP_CONV` $c\ t$, in bottom-up order.

Depending on c, either `TOPDN_CONV (REPEAT1C` c`)` or `REPEAT1C (TOPDN_CONV (REPEAT1C` c`))` may be conversions which achieve more (but are slower) than `TOPDN_CONV` c.

The functions `CONV_GOAL_TAC : conv -> int -> tactic` and `CONV_GOALS_TAC : conv -> tactic` turn a conversion into tactics which will alter a given subgoal, or all subgoals respectively. The function `CONV_RULE : conv -> thm -> thm option` uses a conversion to turn a theorem into a new one.

More such functions and conversionals are available in the file `conv.ML`.

3.2 Ex_eq_conv

Given a term of the form $\exists x\ y\ z.\ P\ x\ y\ z \wedge y = f\ x\ z \wedge Q\ x\ y\ z$ clearly the only possible solution for y is $y = f\ x\ z$. Therefore the term is equivalent to $\exists x\ z.\ P\ x\ (f\ x\ z)\ z \wedge Q\ x\ (f\ x\ z)\ z$.

The conversion `ex_eq_conv'` actually converts the original term to $\exists x\ y\ z.\ P\ x\ (f\ x\ z)\ z \wedge f\ x\ z = f\ x\ z \wedge Q\ x\ (f\ x\ z)\ z$ (i.e, converting y in the body of the original to $f\ x\ z$); `Simp_tac` will simplify this to $\exists x\ z.\ P\ x\ (f\ x\ z)\ z \wedge Q\ x\ (f\ x\ z)\ z$.

Given a term such as $\exists x.\ P\ x \wedge (\exists y.\ x = a + y) \wedge Q\ x$ we can rewrite it to move existential quantifiers outwards, to make `ex_eq_conv'` applicable. (Note that default simplification will do the opposite, i.e. move quantifiers inwards where possible).

3.3 Ex_mono_conv

Given a term of the form $\exists x.\ P\ x\ x$, where the notation $P\ x\ x$ implies that x appears in P in two (or more) places, and where P is monotonic in both (or all) of those arguments, then the term $\exists x_1\ x_2.\ P\ x_1\ x_2$ is equivalent. This is because, given any solution x_1, x_2 for the latter, $x = \max(x_1, x_2)$ will suffice as a solution for the former.

The conversion `ex_mono_conv'` performs this conversion. The resulting term may well not seem "simpler" than the original, but it can allow further simplification. For example, $\exists x.\ a < x * b \wedge c \leq x$ can be converted to $\exists x_1\ x_2.\ a < x_1 * b \wedge c \leq x_2$, and thence (by default Isabelle simplification) to $(\exists x_1.\ a < x_1 * b) \wedge (\exists x_2.\ c \leq x_2)$; then the second conjunct can be simplified away.

3.4 Ex_rel_conv

The conversion `ex_rel_conv'` converts a term such as $\exists x\ y.\ P\ y \wedge b < x - y$ to $\exists x\ y.\ P\ y \wedge \mathit{True}$. It depends on establishing that the conjunct $b < x - y$ can be solved for x.

The result in the example can be simplified to $\exists y.\ P\ y$. It works where there are several existential quantifiers together (as the example given above) and multiple conjuncts. As in the case of `ex_eq_conv'`, it may also be useful to first rewrite so as to gather the existential quantifiers together outside a set of conjuncts.

3.5 Ex_rm_conv

The conversion `ex_rm_conv` is used to convert a term such as $\exists y.\ x \leq y \wedge P\ y$, where P is antitonic in y, to $P\ x$. The reasoning behind this is that $y = x$ is the unique minimum possible solution to the first conjunct for y, and since P is antitonic in y, $P\ x$ holds if and only if $P\ y$ holds for some y such that $x \leq y$.

4 Conclusion

We have shown how the use of Lax Logic to handle constraints and of the Isabelle theorem prover to perform proofs can achieve the "clean yet sound separation" ([4, §4]) of reasoning logically about the properties which "ought" to hold and calculating (in the theorem prover) the constraints under which those properties actually do hold.

We have shown also described a number of conversions suitable for simplifying these automatically generated constraints in the case of the timing constraints for hardware. Although the automatically generated constraints can be complex, the conversions we have described are powerful tools for simplifying them, and we suggest that they would be generally useful for constraints calculated using our method.

Our approach is motivated by practical considerations and does not aim for complete generality. An alternative and more general approach would be to apply Norrish's work on implementing Cooper's algorithm in HOL [10]. This algorithm is a decision procedure for Presburger Arithmetic and relies on a method of transforming a formula into a quantifier-free normal form; since our focus in constraint analysis is not merely on *proving* constraints (that is, on showing they are redundant) but more generally in *simplifying* them, it is the construction of normal forms that interests us. In the above example of the latch, the final form of the constraint is indeed quantifier-free, and considerably shorter than the constraint initially generated. However, there are instances where quantifier elimination would greatly increase the size of a constraint, and in those cases a more compact formulation involving quantifiers might be more intelligible and therefore preferable.

Our constraint-based approach to machine-assisted reasoning would provide a wealth of examples that could be used to compare these two approaches.

References

1. Nick Benton, G.M. Bierman & Valeria de Paiva. Computational Types from a Logical Perspective I. Journal of Functional Programming, 8 (1993), 177–193.
2. H. B. Curry. The elimination theorem when modality is present. Journal of Symbolic Logic, 17 (1952), 249–265.
3. M. Fairtlough, M. Mendler. Propositional Lax Logic. Information and Computation, Vol.137, No.1 (1997), pp. 1–33.
4. M. Fairtlough, M. Mendler, X. Cheng. Abstraction and refinement in Higher-order Logic. Proc. Theorem Proving in Higher-Order Logics 2001. LNCS 2152, 201–216.
5. The HOL System Description, Version 2, Computer Laboratory, Cambridge, 1991.
6. David J King and Philip Wadler. Combining monads. Glasgow Workshop on Functional Programming, Springer Verlag Workshops in Computing Series, Ayr, July 1992.
7. M. Mendler, A Modal Logic for Handling Behavioural Constraints in Formal Hardware Verification. PhD thesis, Edinburgh University, Dept of Computer Science, ECS-LFCS-93-255, 1993.
8. M. Mendler, M. Fairtlough. Ternary simulation: Refinement of binary functions or abstraction of real-time behaviour? Proc. 3rd Workshop on Designing Correct Circuits (DCC'96), Sweden. Baastad, Sweden (M. Sheeran and S. Singh, Eds.), Springer Electronic Workshops in Computing, September, 1996.
9. E. Moggi. Notions of computation and monads, Information and Computation, Vol.93 (1991), pp. 55–92.
10. M. Norrish. Slides on implementing Cooper's algorithm for deciding Presburger arithmetic. `http://www.cl.cam.ac.uk/users/mn200/research/cooper-slides.ps.gz`

Automating Fraenkel-Mostowski Syntax*

Murdoch J. Gabbay[1]

Computer Laboratory, Cambridge University, UK
mjg1003@cl.cam.ac.uk
http://cl.cam.ac.uk/~mjg1003

Abstract. Work with Pitts and others has led to FM (Fraenkel-Mostowski) theory, a fresh understanding of modelling syntax in the presence of variable binding. We discuss the design and other issues encountered implementing these techniques in the mechanised theorem-prover Isabelle.

1 Introduction

It is easy to declare a naïve datatype of terms of some language, for example the untyped λ-calculus,

$$\Lambda = \mu X.\mathrm{Var}\ \mathbf{of}\ \mathbf{Nat} + \mathrm{App}\ \mathbf{of}\ X \times X + \mathrm{Lam}\ \mathbf{of}\ \mathbf{Nat} \times X \tag{1}$$

where **Nat** is the natural numbers. Problems famously arise defining program transformations in the presence of variable binding. For example a substitution function $[t/a]s$ on Λ above should avoid "accidental variable capture" in $[\mathrm{Var}(1)/\mathrm{Var}(0)]s$ for $s = \mathrm{Lam}(1, \mathrm{Var}(0))$. Thus we rename 1 in s to some $i \neq 0, 1$, but then $\mathrm{Var}(i)$ is no longer syntactically a subterm of s and we have made an arbitrary choice about the value of i. The former point causes difficulty with structural induction, the latter because we may have to formally prove irrelevance of the choice made.[1]

All this we could do without, especially in the unforgiving structure of a computer proof assistant such as Isabelle, HOL98, or COQ, or even programming in some language with datatypes. There is much research in this area, for example explicit substitutions ([2]), de Bruijn indices ([3]), and HOAS ([11], [4], [9]).

FM theories are another approach with a pleasingly elementary mathematical foundation. See [7] (my thesis), [5] and [6] (set theory), [8] (higher-order logic), [13] (programming languages), [12] (first-order logic). The label "Fraenkel-Mostowski" honours the creators of set theories designed to prove the independence of the axiom of choice, see [15]: a very special Fraenkel-Mostowski set theory was the first FM theory in the sense of this paper to be created.

In this paper we discuss principles of formally implementing a theory of FM syntax, based on experience doing so in Isabelle [14].

The first design decision of the implementation is the choice of system, Isabelle. We chose Isabelle for its paradigm of constructing arbitrary useable theories (Isabelle/Pure/FOL, Isabelle/Pure/HOL, Isabelle/Pure/CCL, ..., see [14]) in a fixed weak meta-language Isabelle/Pure. This meta-language is a very weak higher-order logic (HOL) containing little more than modus ponens, but to which we may add new types, constants of those types, and axioms on those constants. Thus we may axiomatise a theory in Isabelle/Pure and then work inside that theory. This is good for prototyping a new foundational system such as FM.

2 FM

'FM' may differ depending on whether we do computation or logic. For example compare the typed λ-calculus (a theory of computable functions) to higher-order logic (a theory of all functions). This paper is about logic. FM in computation (programming languages, unification) is under development, see [13, 1].

* The author gratefully acknowledges the funding of UK EPSRC grant GR/R07615 and thanks Andrew Pitts for his suggestions for improvements.

[1] Cf. the work of McKinna and Pollack in the LEGO system, e.g. [10]. FM is quite different but sometimes echoes this work.

'FM' is a set of techniques for α-equivalence with inductive definitions and not a particular theory. We shall now present FM in the style of a higher-order logic. This is not an axiomatic presentation (see [8]) but a 'sketch of salient features, in the style of higher-order logic'. First, three preliminary remarks:

1 (Types). We shall write type annotations in two styles: $x : \alpha$ and x^α both mean "x of type α". $\diamond$

2 (HOL sets). Higher-order logic has a notion of set, where 'α-sets' is predicates $\alpha \to \mathtt{Bool}$ also written $\mathcal{P}(\alpha)$. We borrow set notation, for example writing $x \in X$ for '$(X\ x)$', $X \subseteq Y$ for '$\forall x.\ (X\ x) \to (Y\ x)$', and $\emptyset$ for $\lambda x.\bot$ and α for $\lambda x^\alpha.\top$. $\diamond$

3 (Meaning of infinite). In FM theories not all types can be well-ordered (bijected with an ordinal, see [8, Lemma 4.10(5)]). Therefore, a reading of 'X is infinite' as $X \cong \mathbb{N}$ is suspect. In FM we use '$X \notin \mathcal{P}_{fin}(X)$' where $\mathcal{P}_{fin}(X)$ is the inductively defined type of finite subsets of X. $\diamond$

An FM theory has:

4 (Atoms). An infinite type of ***atoms*** $a, b, c, \ldots : \mathbb{A}$ to model variable names. For example in an inductively defined type of expressions for types,

$$\Sigma ::= \mathbf{TypeVar\ of\ } \mathbb{A} + \mathbf{Product\ of\ } \Sigma \times \Sigma + \mathbf{DisjSum\ of\ } \Sigma \times \Sigma, \tag{2}$$

type variables are represented as $\mathbf{TypeVar}(a)$ for $a : \mathbb{A}$. $\diamond$

5 (Transposition). There is a (polymorphically indexed class of) constant(s)

$$\mathbf{Tran} : \mathbb{A} \to \mathbb{A} \to \sigma \to \sigma, \tag{3}$$

read "***transposition***". Write $(\mathbf{Tran}\, a\, b\, x^\sigma)$ as $(a\ b).x$. The intuitive meaning of $(a\ b).x$ is as transposing a and b in x. For example if $x = \langle a, b\rangle$ then $(a\ b).x$ should equal $\langle b, a\rangle$. This is made formal by the following equational axioms which **Tran** must satisfy, and equivariance below:

$$(a\ a).x = x \tag{4}$$

$$(a\ b).(a\ b).x = x \tag{5}$$

$$(a\ b).(c\ d).x = (c\ d).\ ((c\ d).a\ (c\ d).b)\ .x \tag{6}$$

$$(a\ b).n^{\mathbb{A}} = if(n = a, b, if(n = b, a, n)) \tag{7}$$

where $if(test, t_1, t_2)$ is Isabelle-like notation meaning "if $test$ then t_1 else t_2". $\diamond$

6 (Equivariance). For a term f with free variables $x_1, \ldots, x_n$,

$$(a\ b).f(x_1, \ldots, x_n) = f((a\ b).x_1, \ldots, (a\ b).x_n). \tag{8}$$

In the case that f has no free variables we have the special case that $(a\ b).f = f$.

We say the language is ***equivariant***. An ***equivariant element*** x is one such that for all a, b, $(a\ b).x = x$. From (8) for $n = 0$ it follows that closed terms denote equivariant elements. $\diamond$

Definition 7 (Smallness, И). *Write $\mathcal{P}_{fin}(\mathbb{A})$ for the* HOL *set of finite subsets of $\mathbb{A}$. Say a set $X \subseteq \mathbb{A}$ is cofinite when its complement $\mathbb{A} \setminus X$ is finite. Write $\mathcal{P}_{cofin}(\mathbb{A})$ for the* HOL *set of cofinite subsets of $\mathbb{A}$. For $P : \mathbb{A} \to \mathtt{Bool}$ write 'ИP' or 'И$a.\ P(a)$' for $P \in \mathcal{P}_{cofin}(\mathbb{A})$.*

$\mathbb{A}$ is infinite from remark 4 so we can read И$a.\ P(a)$ as "for all but finitely many $a : \mathbb{A}$, $P(a)$", or more loosely as "for *most* $a : \mathbb{A}$, $P(a)$". We may call finite $P : \mathbb{A} \to \mathtt{Bool}$ ***small*** and their complements, cofinite sets, ***large***. Thus P is large precisely when И$a.\ P(a)$, and small precisely when И$a.\ \neg P(a)$.

Definition 8. *Define $a\#x \overset{\text{def}}{=}$ (И$b.\ (b\ a).x = x$) and read this as "a is not 'in' x" or "a is apart from x". The intuition is that, since transposition transposes b for a in x and since b is fresh, if $(b\ a).x = x$ then certainly a is not in x.*

We have an axiom stating that 'most' atoms are not 'in' $x : \sigma$:

$$\text{И}a.\ a\#x. \tag{Small}$$

Expanding definition 8 this becomes $\text{И}a.\ \text{И}b.\ (a\ b).x = x$. Write

$$\mathbf{Supp}(x) \stackrel{\text{def}}{=} \{a : \mathbb{A} \mid \neg a\#x\}$$

Then (Small) is equivalent to

$$\mathbf{Supp}(x) \in \mathcal{P}_{fin}(x) \tag{9}$$

and we can also read (Small) as "x ***has finite support***".

9 (Some observations). 'In' does *not* correspond to (HOL-)set membership. For example,

$$n \notin L = \mathbb{A} \setminus \{n\} \quad \textbf{but} \quad n \in \mathbf{Supp}(L).$$

We might think of $\mathbf{Supp}(x)$ as an object-level notion of those atoms occurring in some *meta-level* term which x denotes.

Datatypes of syntax T certainly satisfy (9). Terms $t : T$ are finite[2] so mention only finitely many atoms, and cofinitely many $a : \mathbb{A}$ satisfy $a\#t$. ◇

10 (И excellent properties). Higher types such as $\mathbb{A} \to \texttt{Bool}$ also satisfy (9). Observe that (using some sets notation)

$$P : \mathbb{A} \to \texttt{Bool} = \{x \mid P(x)\}.$$

It follows from (8) that

$$(a\ b).P = \{(a\ b).x \mid P(x)\}.$$

We can verify by calculation that $(a\ b).P = P$ if and only if $a, b \in P$ or $a, b \notin P$. When we combine this with (9) it follows that either 'most' atoms are in P or most are not in P:

$$\mathcal{P}(\mathbb{A}) \cong \mathcal{P}_{fin}(\mathbb{A}) + \mathcal{P}_{cofin}(\mathbb{A}). \tag{10}$$

We can rewrite this as $\neg\text{И}a.\ P(a) \Leftrightarrow \text{И}a.\ \neg P(a)$. Now the full set $\lambda x.\top = \mathbb{A} \subseteq \mathbb{A}$ is clearly cofinite so $\text{И}a.\ \top = \top$. Combining this with other properties of cofinite and finite sets we obtain the algebraic commutativity properties:

$$
\begin{aligned}
\text{И}a.\ P(a) \wedge \text{И}a.\ Q(a) &\Longleftrightarrow \text{И}a.\ P(a) \wedge Q(a) && (11)\\
\text{И}a.\ P(a) \vee \text{И}a.\ Q(a) &\Longleftrightarrow \text{И}a.\ P(a) \vee Q(a) && (12)\\
\text{И}a.\ \neg P(a) &\Longleftrightarrow \neg\text{И}a.\ P(a) && (13)\\
& \quad\ \ \text{И}a.\ \top && (14)\\
& \quad\ \ \neg\text{И}a.\ \bot && (15)\\
(\text{И}a.\ P(a)) \wedge Q &\Longleftrightarrow \text{И}a.\ (P(a) \wedge Q), && (16)
\end{aligned}
$$

that is, И distributes over $\wedge$, $\vee$, $\to$, $\neg$, $\top$ and $\bot$. These strong properties make И convenient to work with in a mechanised context. They also place И in an interestingly 'in between' $\forall$ and $\exists$, the equations being informally:

$$\neg\forall\neg = \exists \qquad \neg\text{И}\neg = \text{И} \qquad \neg\exists\neg = \forall.$$

◇

[2] Extending FM to infinitary syntax is possible and interesting.

3 α-equivalence on a simple datatype

11. Tran is how an FM theory renames object-level variables. It interacts with object-level syntax better than atom-substitution $[b/a]$. For example $[b/a]$ applied to $t = \mathbf{Lam}(a).\mathbf{Lam}(b).a(b)$ raises all the usual problems with capture avoidance, whereas $(b\ a).t = \mathbf{Lam}(a).\mathbf{Lam}(b).b(a)$ is α-equivalent to t itself. Similarly, $(b\ a).(\mathbb{A} \setminus \{a\}) = \mathbb{A} \setminus \{b\}$ wheras $[b/a](\mathbb{A} \setminus \{a\}) = \mathbb{A} \setminus \{a\}$. ◇

We can define an α-equivalence relation $=_\alpha$ by cases on inductive types. For example:

Definition 12.

$$\begin{array}{lcll} L & \stackrel{\text{def}}{=} & \textit{\textbf{TyVar of}}\ \mathbb{A} + \textit{\textbf{TyProd of}}\ L \times L + \textit{\textbf{TyAbs of}}\ \mathbb{A} \times L & \\ =_\alpha & \stackrel{\text{def}}{=} & \textit{\textbf{TyVar}}(a) =_\alpha \textit{\textbf{TyVar}}(b) & \leftarrow a = b \\ & & \textit{\textbf{TyProd}}(t_1, t_2) =_\alpha \textit{\textbf{TyProd}}(t'_1, t'_2) & \leftarrow t_1 =_\alpha t'_1 \wedge t_2 =_\alpha t'_2 \\ & & \textit{\textbf{TyAbs}}(a.t) =_\alpha \textit{\textbf{TyAbs}}(a'.t') & \leftarrow \text{И}b.\ (b\ a).t =_\alpha (b\ a').t' \end{array}$$

Here L is intended to be a type of expressions for types. The definition of L above might be written in more familiar style as

$$l ::= \mathbf{TyVar}(a) \mid \mathbf{TyProd}(l, l) \mid \mathbf{TyAbs}(a.l) \qquad a : \mathbb{A},$$

and sugared to (writing σ for l a type and α for $a : \mathbb{A}$ a type variable)

$$\sigma ::= \alpha \mid \sigma \times \sigma \mid \bigwedge \alpha.\sigma.$$

We shall use L, $=_\alpha$, and $=_\alpha{}'$ defined below in (18), as a running object of study in the rest of this paper. In the rest of this section and elsewhere the proofs given are semi-formal accounts of the formal proofs as they might be conducted in Isabelle.

This machinery allows us to quite easily prove some nice properties for $=_\alpha$, for example transitivity:

Lemma 13. $=_\alpha$ is transitive.

Proof. By induction on syntax using hypothesis

$$\phi(t_1) \quad \stackrel{\text{def}}{\iff} \quad \forall t_2, t_3.\ (t_1 =_\alpha t_2 \wedge t_2 =_\alpha t_3) \to t_1 =_\alpha t_3.$$

The significant case is of $t_1 = \mathbf{TyAbs}(a_1, t'_1)$. So suppose $\phi(t'_1)$, $t_1 =_\alpha t_2$, and $t_2 =_\alpha t_3$. Then $t_2 = \mathbf{TyAbs}(a_2, t'_2)$ and $t_3 = \mathbf{TyAbs}(a_3, t'_3)$, and

$$(\text{И}b.\ (b\ a_1).t'_1 =_\alpha (b\ a_2).t'_2) \wedge (\text{И}b.\ (b\ a_2).t'_2 =_\alpha (b\ a_3).t'_3).$$

We now equationally apply (11) to deduce

$$\text{И}b.\ (b\ a_1).t'_1 =_\alpha (b\ a_2).t'_2 =_\alpha (b\ a_3).t'_3. \tag{17}$$

Now we assumed $\phi(t'_1)$, not $\phi((b\ a_1).t'_1)$. But we can apply equivariance (8) to $\phi(x)$ to deduce $\phi(t'_1) \iff \phi((b\ a_1).t'_1)$, which allows us to complete the proof. □

We can also define a more traditional α-equivalence $=_\alpha{}'$:

$$\begin{array}{ll} \mathbf{TyVar}(a) =_\alpha{}' \mathbf{TyVar}(b) & \leftarrow a = b \\ \mathbf{TyProd}(t_1, t_2) =_\alpha{}' \mathbf{TyProd}(t'_1, t'_2) & \leftarrow t_1 =_\alpha{}' t'_1 \wedge t_2 =_\alpha{}' t'_2 \\ \mathbf{TyAbs}(a, t) =_\alpha{}' \mathbf{TyAbs}(a', t') & \leftarrow \exists b.\ [b/a].t =_\alpha{}' [b/a']t' \wedge \\ & \qquad b \notin n(t) \cup n(t') \cup \{a, a'\} \end{array} \tag{18}$$

in terms of an inductively defined names-of function $n(t)$

$$\begin{array}{ll} n(\mathbf{TyVar}(a)) & = \{a\} \\ n(\mathbf{TyProd}(t_1, t_2)) & = n(t_1) \cup n(t_2) \\ n(\mathbf{TyAbs}(a, t)) & = \{a\} \cup n(t) \end{array} \tag{19}$$

and an inductively defined atom-for-atom substitution function

$$\begin{aligned}
[b/a]\mathbf{TyVar}(a) &= \mathbf{TyVar}(b) \\
[b/a]\mathbf{TyVar}(n) &= \mathbf{TyVar}(n) \quad n \neq a \\
[b/a]\mathbf{TyProd}(t_1, t_2) &= \mathbf{TyProd}([b/a]t_1, [b/a]t_2) \\
[b/a]\mathbf{TyAbs}(n, t) &= \mathbf{TyAbs}([b/a]n, [b/a]t).
\end{aligned} \tag{20}$$

$n(t)$ and $[b/a]t$ are simple and make no allowance for free variables or capture-avoidance, but they suffice for our needs.

Suppose we want to prove $t_1 =_\alpha t_2 \leftrightarrow t_1 =_\alpha{}' t_2$. A pleasing and clean method would be to prove

$$\text{И}b.\ [b/a]t = (b\ a).t \tag{21}$$

$$\exists b.\ [b/a].t =_\alpha{}' [b/a']t' \ \wedge\ b \notin n(t) \cup n(t') \cup \{a, a'\} \iff \tag{22}$$
$$\text{И}b.\ [b/a]t =_\alpha{}' [b/a']t'.$$

Proof (of (21)*).* We can use structural induction for a fixed with hypothesis ϕ

$$\phi(t) \overset{\text{def}}{\iff} \text{И}b.\ [b/a]t = (b\ a).t.$$

Suppose $t = \mathbf{TyProd}(t_1, t_2)$. By definition from (20), $[b/a]\mathbf{TyProd}(t_1, t_2) = \mathbf{TyProd}([b/a]t_1, [b/a]t_2)$, and by equivariance (8), $(b\ a).\mathbf{TyProd}(t_1, t_2) = \mathbf{TyProd}((b\ a).t_1, (b\ a).t_2)$. By hypothesis we know

$$(\text{И}b.\ [b/a]t_1 = (b\ a).t_1) \wedge (\text{И}b.\ [b/a]t_2 = (b\ a).t_2).$$

By (11) and applying the equalities under **TyProd** we obtain the result.

The cases of **TyVar** and **TyAbs** are no different. Each time, equivariance of $(b\ a)$ as illustrated in (8) allows us to push transposition down through the structure of a term and replicate the inductive behaviour of $[b/a]$. This is a general pattern. □

Note from this proof how transposition with equivariance has provided a 'general axiomatic theory of (purely inductive) renaming'.

Proof (of (22)*).* The proof of (22) is rather more involved. It is best to work from the following lemmas:

$$n(t) \in \mathcal{P}_{fin}(\mathbb{A}) \tag{23}$$
$$X \in \mathcal{P}_{fin}(\mathbb{A}) \implies (b \notin X \leftrightarrow b\#X) \tag{24}$$
$$b \notin n(t) \iff b\#t \tag{25}$$
$$b\#x \wedge b\#f \implies b\#f(x) \tag{26}$$
$$b\#(f(x)) \wedge b\#f \wedge f\ \textbf{injective} \implies b\#x \tag{27}$$
$$b\#c \quad c\ \textbf{a closed term} \tag{28}$$
$$b\#P^{\mathbb{A}\to\mathsf{Bool}} \wedge P(b) \implies \text{И}b.\ P(b) \tag{29}$$
$$\text{И}b.\ P(b) \implies \exists b.\ b\#P \wedge P(b) \tag{30}$$
$$\text{И}b.\ P(b) \implies \forall b.\ b\#P \implies P(b) \tag{31}$$
$$\exists b.\ b\#x \tag{32}$$

The proof now proceeds as follows. We must prove

$$\exists b.\ [b/a]t =_\alpha{}' [b/a']t' \wedge b \notin n(t) \cup n(t') \cup \{a, a'\} \iff \text{И}b.\ [b/a]t =_\alpha{}' [b/a']t'.$$

Write $P \overset{\text{def}}{=} \lambda a, a', t, t'.\lambda b.[b/a]t =_\alpha{}' [b/a']t'$ and use (25) (proved from (23) and (24)) and to rewrite this to

$$\exists b.\ b\#t, t', a, a' \wedge P(t, t', a, a', b) \iff \text{И}b.\ P(t, t', a, a', b),$$

where $b\#x_1,\ldots,x_n$ denotes the conjunction $\bigwedge_i b\#x_i$.
Left-right implication. We must prove

$$\exists b.\ b\#t,t',a,a' \wedge P(t,t',a,a',b) \implies \text{И}b.\ P(t,t',a,a',b).$$

We eliminate the existential quantifier and obtain

$$b\#t,t',a,a' \wedge P(t,t',a,a',b) \implies \text{И}b.\ P(t,t',a,a',b). \tag{33}$$

We resolve against (29) to obtain

$$b\#t,t',a,a' \wedge P(t,t',a,a')(b) \implies b\#P(t,t',a,a') \wedge P(t,t',a,a')(b).$$

which simplifies to $b\#t,t',a,a' \implies b\#P(t,t',a,a')$. We repeatedly resolve against (26) to reduce to $b\#P$, and finish this off with (28).
Right-to-left implication. We must prove

$$\text{И}b.\ P(t,t',a,a',b) \implies \exists b.\ b\#t,t',a,a' \wedge P(t,t',a,a',b).$$

Now here we have a problem. Clearly we would like to eliminate И using (30) to obtain

$$b\#P(t,t',a,a') \wedge P(t,t',a,a',b) \implies \exists b.\ b\#t,t',a,a' \wedge P(t,t',a,a',b),$$

identify the b in the conclusion with the b in the hypotheses, and simplify. But we obtain

$$b\#P(t,t',a,a') \implies b\#t,t',a,a'.$$

This implication does not follow for general P, nor even for our particular P: if P were injective we could apply (27) repeatedly, but it is a predicate mapping into `Bool` and is not injective.

However we can use (32) to introduce into the context some b fresh for *any* x, so instantiate x to the 3-tuple

$$<n(t),n(t'),\{a,a'\}>. \tag{34}$$

Now we can apply (27) repeatedly to obtain

$$b\#t,t',a,a' \wedge \text{И}b.\ P(t,t',a,a',b) \implies \exists b.\ b\#t,t',a,a' \wedge P(t,t',a,a',b).$$

(Here there is also a hypothesis $b\#\lambda x_1,x_2,x_3.<x_1,x_2,x_3>$ but this gives us no information since we get it for free from (28), so we drop it.) This simplifies to

$$b\#t,t',a,a' \wedge \text{И}b.\ P(t,t',a,a',b) \implies P(t,t',a,a',b).$$

But now we have another problem. If we eliminate И using (30) we obtain for a variable symbol b',

$$b\#t,t',a,a' \wedge b'\#P(t,t',a,a') \wedge P(t,t',a,a',b') \implies P(t,t',a,a',b).$$

We need a different elimination rule for И which does not introduce a new variable into the context, and this is provided by (31), with which we can finish off the proof. □

4 Morals from the proofs

In the previous section we have seen the beginnings of the automated theory of $(a\ b)$, $\#$, introducing a fresh name, and И. We now bring it out explicitly.

14 (Theory of transposition). Given a conclusion of the form $s = (a\ b).t$, use (8) to simplify the RHS by drawing transposition down to the variables on the right hand side. Similarly for other binary predicates such as $\leftrightarrow$ or also $\#$. So for example

$$s = (a\ b).<x,y> \quad \textbf{simplifies to} \quad s = <(a\ b).x,(a\ b).y>.$$

This algorithm can fail, for example on the goal $(a\ b).<x,y> = (a\ b).<x,y>$. Call it **push**, because it 'pushes' transposition into the structure of the term on the right of an equality. In an implementation **push** would denote a tactic. We shall continue to give such names to algorithms which would denote tactics. ◇

15 (Theory of #). Given a goal of the form $a\#t$ repeatedly apply (26) and (28) to simplify it to component parts. So for example

$$a\#{<}x,y{>}$$

reduces to $a\#x \wedge a\#y \wedge a\#\lambda x.y.{<}x,y{>}$, and then to $a\#x \wedge a\#y$. This algorithm can also fail, for example in $a\#\pi_1{<}\top,a{>}$ we should perform β-reduction first, otherwise we finish up with $a\#a$, which is untrue. Call the algorithm `split#`. $\diamond$

Inductive proof on inductive types can, with proper handling and properly coordinated automated procedures, be made to produce very uniform proof-obligations which are amenable to this kind of treatment, with only slightly more sophisticated algorithms.

16 (Introducing a fresh name). (32) allows us to introduce a new variable b into the context, fresh for x for any x: given the proof-state

$$\forall x_1,\ldots,x_n.\ Conds(x_1,\ldots,x_n) \implies Concl(x_1,\ldots,x_n)$$

we can reduce to

$$\forall x_1,\ldots,x_n,b.\ Conds(x_1,\ldots,x_n) \wedge b\#t(x_1,\ldots,x_n,b) \implies Concl(x_1,\ldots,x_n)$$

for any t.

We can now take t to be the n-tuple ${<}\mathbf{Supp}(x_1),\ldots,\mathbf{Supp}(x_n){>}$. Repeated applications of (27) reduce $b\#t$ to $\bigwedge_i b\#\mathbf{Supp}(x_i)$. It is a lemma that $b\#\mathbf{Supp}(u) \iff b\#u$, so we obtain

$$\forall x_1,\ldots,x_n,b.\ Conds(x_1,\ldots,x_n) \wedge \bigwedge_i b\#x_i \implies Concl(x_1,\ldots,x_n).$$

In other words, "we can always invent a fresh b". We applied this technique ad-hoc in (34). Call the algorithm `newname`. $\diamond$

17 (Theory of И). The treatment of И is more complex. There are two broad styles of reasoning on И, equational reasoning using for example properties such as (11) and (16), and directed reasoning using intro- and elim- rules such as (29), (30), and (31). Both are useful. For example equations in (17), and intro- and elim- rules in the proof of (22).

A further complication of the treatment of intro- and elim- rules is that И seems to have two pairs of them. In full, they are

$$\exists b.\ (b\#P^{\mathbb{A}\to\mathtt{Bool}} \wedge P(b)) \implies \text{И}b.\ P(b) \tag{35}$$
$$\text{И}b.\ P(b) \implies (\exists b.\ b\#P \wedge P(b)) \tag{36}$$
$$\forall b.\ (b\#P^{\mathbb{A}\to\mathtt{Bool}} \to P(b)) \implies \text{И}b.\ P(b) \tag{37}$$
$$\text{И}b.\ P(b) \implies (\forall b.\ b\#P \implies P(b)). \tag{38}$$

For practical purposes these pair off naturally as (35) with (38) and (37) with (36). The first pair requires we find in the context a fresh b. The second pair introduces that fresh b, but only fresh for P. We can do better than this using (32) as in remark 16, so this latter pair seems less useful.

The complete algorithm is therefore: simplify using (11) to collect all И quantifiers in the hypotheses into one single quantifier. Also use (13) to draw negations under the И quantifier. Finally, apply the intro- and elim- rules (35) with (38), possibly augmented with remark 16 to generate a fresh name where necessary. Thus for example

$$\forall params.\ \text{И}a.\ P(a) \wedge \neg\text{И}a.\ Q(a) \implies \neg\text{И}a.\ R(a)$$

simplifies to

$$\forall params.\ \text{И}a.\ P(a) \wedge \neg Q(a) \implies \text{И}a.\ \neg R(a),$$

a fresh b is introduced

$$\forall params, b.\ b\#params \wedge \text{И}a.\ P(a) \wedge \neg Q(a) \implies \text{И}a.\ \neg R(a),$$

the intro- and elim- rules reduce this to

$$\forall params, b.\ P(b) \wedge \neg Q(b) \implies \neg R(b),$$

and proof-search proceeds as normal. In the case that the fresh b is already in the context, as happened in (33), we use that supplied b instead. Call this algorithm `newsimp`. $\diamond$

5 Difficulties implementing the algorithms

There are many technical difficulties putting the ideas of section 4 into practice.

18 (`split#`). `split#` is described in remark 15. The steps of the algorithm are:

1. repeated resolution with (26) followed by, when this fails.
2. resolution with (28).

There are difficulties with both steps.

1. Isabelle resolution with Isabelle unification is higher-order. $a\#f(x)$ unifies with a goal $a\#t$ for x matches t and f matches $\lambda x.x$ the identity, and we have a non-terminating loop. The solution is to write ML code to only allow this step when t is syntactically an application term `t1 $ t2`, and package this up as an Isabelle wrapper. An Isabelle wrapper, simplistically put, is an Isabelle theorem 'wrapped' in ML code which provides some intelligent control on how it may be applied. see remark 20.
2. It is impossible at object level to decide whether a term of the meta-level is closed or not. Again, we need an ML wrapper.

The algorithm `push` described in remark 14 is similar and also requires wrappers. ◇

19 (`newname`). To introduce a fresh b fresh for all variables $x_1, \ldots, x_n$ in the context, as we saw in remark 16, we must examine those names. This is, as in the previous remark, an operation on the meta-level syntax and must be implemented by an ML wrapper which examines that syntax. ◇

20 (Isabelle wrappers). We observed in remarks 19 and 18 that three significant FM features require ML wrappers in implementation (`split#`, `push`, and `newname`).

Isabelle proof proceeds imperatively by applying tactics to a proof-state. Simple tactics may apply a particular transformation to the state. More complex tactics will carry out some kind of proof-search. These automated tactics (written in ML) give Isabelle proving much of its power. They are all essentially tree-search algorithms of various kinds based on a library of Isabelle theorems which may be equalities, intro-rules, elim-rules, as the case may be. In inductive reasoning we use this automation to automatically handle the dozens if not hundreds and thousands of separate cases which a proof may entail. Wrappers are applied in between proof-steps and perform well as intelligent agents which may examine the way the proof-state is developing and perform for example some kind of garbage-collection.

But consider the example of `split#`. This is implemented as a wrapper as discussed above in remark 18 but morally it is clearly a pair of intro-resolution rules:

$$a\#f \wedge a\#x \implies a\#fx \quad \textbf{and} \quad a\#c \quad \textbf{if } c \textbf{ closed.}$$

In proof-search however `split#` will only be applied if none of the standard Isabelle theorems is applicable. We cannot, using wrappers, interleave it 'horizontally' with the standard Isabelle theorems, only 'vertically' with lower precedence, and in consequence proof-search is inefficient. Unfortunately there seems no cure other than dedicated FM proof-search ML code, or to hack existing code to hardwire algorithms such as `split#`, `push`, and `newsimp`. ◇

Now consider our treatment of the logic of И. This consists of equational theory such as (11) and (16), of intro- and elim- rules

$$a\#P,\ P(a) \implies \text{И}a.\ P(a) \quad \textbf{and} \quad (\text{И}a.\ P(a)),\ a\#P \implies P(a),$$

and of `newname` discussed in remark 17.

In this and in the equations immediately following we introduce two items of notation. $\bigwedge$ here is not a conjunction (as previously used written $\bigwedge_i prop_i$) but a meta-level Isabelle/Pure universal quantification ($\bigwedge x.\ prop(x)$). Also, a comma , denotes meta-level conjunction. I shall not be completely strict about distinguishing meta-level Pure from object-level HOL, but $\bigwedge$ and , where used will definitely denote the former.

As a simple example of a proof involving И consider a proof of

$$\bigwedge P.Q.\ \text{И}a.\ P(a),\ \text{И}a.\ Q(a) \implies \text{И}a.\ P(a) \wedge Q(a). \tag{39}$$

Inductive reasoning tends to be resolution-based, so we prefer an algorithm in that style. Accordingly we apply the intro- and elim- rules above, along with the conjunction intro-rule $A, B \Rightarrow A \wedge B$, to obtain

$$\begin{aligned}
\bigwedge P, Q.\ P(?a(P,Q)), Q(?b(P,Q)) &\implies ?a(P,Q)\#P\\
\bigwedge P, Q.\ P(?a(P,Q)), Q(?b(P,Q)) &\implies ?a(P,Q)\#Q\\
\bigwedge P, Q.\ P(?a(P,Q)), Q(?b(P,Q)) &\implies P(?a(P,Q))\\
\bigwedge P, Q.\ P(?a(P,Q)), Q(?b(P,Q)) &\implies Q(?b(P,Q)).
\end{aligned}$$

Here $?a(P,Q)$ and $?b(P,Q)$ are unknowns which may be instantiated to any expression with free variables at most P, Q. The two freshness subgoals cannot be proved. We can use `newname` to introduce a fresh parameter into the context, but that only gives us

$$\bigwedge P, Q, b.\ P(?a(P,Q)), Q(?b(P,Q)), b\#P, b\#Q \implies ?a(P,Q)\#P$$

and $?a(P,Q)$ cannot be instantiated to b because b is a new free variable not amongst P, Q. Thus we need to apply `newname` before the resolution steps and then the proof succeeds.

In another situation such as proving (31) the fresh name may be provided by the previous proof-context and we certainly do not want to apply `newname`: it will cause unknowns to be instantiated to an irrelevant fresh parameter. It seems difficult to express a sensible and efficient compromise algorithm for this kind of proof-search.

In the rest of this section we step back and take a high-level view of these problems. In Isabelle and other theorem provers there are actually two kinds of variables. Free variables a, b, x, y and unknowns $?a, ?b, ?x, ?y$. Free variables are 'universal': they have an arbitrary value which ranges over all possible values. Unknowns are 'existential': they should, by the end of the proof-search, be instantiated to some specific term t. With these built into the meta-level of Isabelle/Pure the intro- and elim- rules for universal and existential quantification are easy to write. This need not be the case. For example in second-order λ-calculus existential quantification can be expressed using universal quantification. Theorem provers do not use this because it is nasty to work with in implementation.

It seems that the underlying problem may be that we are trying to encode using both a and $?a$ a kind of 'freshness' variable corresponding to the 'new' quantifier И. The fact that we need both reflects the $\forall/\exists$ duality of И mentioned in remark 10. Like unknowns, $?a$ a freshness variable depends on a context, for which it is fresh, and two sufficiently fresh freshness variables may be assumed equal, 'instantiated to each other', where convenient (think for instance of proving (11) or (12)). Like universals, freshness variables when introduced extend the context, and other terms and variables may depend on them if they are introduced later (e.g. other freshness variables). Trying to usefully express this in a dedicated logic belongs to future work.

6 The technical lemmas

This section can be skipped. For the interested reader we show a simple algorithm in action, constructed using the tactics developed in section 4. The point is that it neatly settles most of (23) to (32), which means we have a decent algorithm. We skip to the fourth one (26) $b\#x \wedge b\#f \implies b\#f(x)$.

Unfold definition 8 and apply `newname`. We obtain

$$\bigwedge b, x, f.\ \text{И}c.\ (c\ b).x, \text{И}c.\ (c\ b).f \implies \text{И}c.\ (c\ b).f(x).$$

Apply `newname`

$$\bigwedge b, x, f, c.\ c\#b, x, f\text{И}c.\ (c\ b).x, \text{И}c.\ (c\ b).f \implies \text{И}c.\ (c\ b).f(x)$$

then `newsimp` to obtain

$$\bigwedge b, x, f, c.\ c\#b, x, f(?c1(b,x,f,c)\ b).x = x, (?c1(b,x,f,c)\ b).f = f$$
$$\implies (?c2(b,x,f,c)\ b).f(x) = f(x).$$

It is now simple to instantiate $?c1(b, x, f, c)$ and $?c2(b, x, f, c)$ to c, but we cannot apply **push** to the conclusion and finish the proof because $(?c2(b, x, f, c)\ b)$ is on the left, not the right. I had elided the following detail: transposition is invertible on each type by (5) so $x = (u_1\ u_2).y \Rightarrow (u_1\ u_2).x = y$. **push** applies this as an intro-rule, to "draw transposition to the right". With this elaboration the proof runs smoothly.

The proof of (27) runs along similar lines. To prove (28) $b\#c$ we unfold definitions and use **newsimp** to obtain

$$\bigwedge b, c.\ (?n(b, c)\ b).c = c.$$

If c is a closed term **push** solves this completely (otherwise proof fails, as it should).

(29), (30), (31), are proved by the same script as (26). In fact, the script also proves (27) though its behaviour for that goal, the path outlined in the previous paragraph, is a little special.

(32) underlies **newname**. The proof is best tailor-made. We rewrite it as $\exists b.\ b\#x \wedge \top$ and intro-resolve against (36) for $P = \lambda b.\top$, we now have $\text{И}b.\ b\#x$, an instance of the axiom (9).

7 The state of the implementation

An Isabelle/FM implementation exists but it is based on set theory rather than higher-order logic. This creates technical difficulties which ultimately proved insurmountable for the following reason. Consider the theorem

$$\mathbf{TyProd}(t_1, t_2) = \mathbf{TyProd}(t'_2, t'_2) \implies t_1 = t'_1.$$

In HOL this is rendered as

$$\mathbf{TyProd}(t_1^L, t_2^L) = \mathbf{TyProd}({t'_2}^L, {t'_2}^L) \implies t_1^L = {t'_1}^L$$

where we include all type annotations. In sets the same theorem is

$$\mathbf{TyProd}(t_1, t_2) = \mathbf{TyProd}(t'_2, t'_2),\ t_1 \in L, t_2 \in L, t'_1 \in L, t'_2 \in L \implies t_1 = t'_1.$$

The difference is that when we intro-resolve against the HOL version we get one subgoal, whereas the sets version produces five (one each for each hypothesis of the implication, which must be established in order to apply it). A sets-based treatment of inductive datatypes overcomes this by implementing **TyProd** by some constructor which is is injective on the entire sets universe "by coincidence", probably **Inr**(**Inl**(−)). In FM this is not possible for various reasons which we now sketch.

Atoms must be marked as belonging to atomic type, the И quantifier introduces fresh variables of atomic type which must be marked as such, and atom-abstraction $a, x \mapsto a.x$ (which we have not discussed in this paper, see [8, Section 6] or [6, Section 5]) is fundamentally non-injective so that the typing conditions can actually get quite complex.

Considerable ingenuity went into minimising the impact of these typing conditions in a sets environment (this should soon be the subject of a technical report). The price of using a HOL environment is precisely its benefit, the relative rigidity the typing gives the theory relative to sets, with both theoretical and practical consequences. In the recently-published [8] we provide what we hope is an elegant solution to the theoretical difficulties which will also be implementable, and it remains to try implementing the approach.

8 Conclusions

This paper has given a very simplified account of the problem of producing an implementation of a new foundational system FM with new and unfamiliar predicates and constructors. We considered two simple examples:

- Some theory of a datatype of types with universal types Σ and relations of α-equivalence for it $=_\alpha$ (defined using FM structure) and $=_\alpha{}'$ (defined in a more traditional style).
- Some technical FM lemmas (23) to (32).

These examples illustrated a fairly rich and representative selection of problems. We presented solutions to these problems and discussed their limitations. Another contribution of this paper is in what it elides: there are complications to automating FM which this paper has tried to bring out, but the short, slick, parts in between are the proof of how far we have already come.

References

1. FreshML homepage. `http://www.cl.cam.ac.uk/~amp12/research/freshml/index.html`.
2. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.
3. H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics (revised ed.)*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.
4. M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science*, pages 193–202. IEEE Computer Society Press, Washington, 1999.
5. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, Washington, 1999.
6. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 2001. Special issue in honour of Rod Burstall. To appear.
7. Murdoch J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence*. PhD thesis, Cambridge, UK, 2000.
8. Murdoch J. Gabbay. FM-HOL, a higher-order theory of names. In *35 Years of Automath*. Heriot-Watt University, Edinburgh, Scotland, April 2002.
9. M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual Symposium on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, Washington, 1999.
10. James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3-4):373–409, 1999.
11. Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, 1988.
12. A. M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer-Verlag, Berlin, 2001.
13. A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.
14. Isabelle Projects. http://www.cl.cam.ac.uk/users/lcp/Isabelle/projects.html.
15. J. Truss. Permutations and the axiom of choice. In H.D.Macpherson R.Kaye, editor, *Automorphisms of first order structures*, pages 131–152. OUP, 1994.

A Formal Correctness Proof of the SPIDER Diagnosis Protocol*

Alfons Geser[1] and Paul S. Miner[2]

[1] ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199, USA
geser@icase.edu
http://www.icase.edu/~geser
[2] Mail Stop 130, NASA Langley Research Center, Hampton, VA 23681-2199, USA
p.s.miner@larc.nasa.gov
http://shemesh.larc.nasa.gov/~psm

Abstract. This paper presents two protocols of SPIDER, a fault tolerant broadcast communication architecture. The Interactive Consistency Protocol ("Byzantine Agreement") takes care of reliable message broadcast in the presence of malicious faults. The Diagnosis Protocol distributes local information about the health status of nodes through the network, such that each node arrives at a correct and consistent classification of which nodes are faulty and which are not. Correctness here means that only faulty nodes are convicted. Such diagnostic information may be useful for on-line maintenance. The two protocols are based on each other: Diagnosis uses the Interactive Consistency Protocol for reliable broadcast of accusation messages. Interactive Consistency relies on up-to-date health status information and produces diagnostic data. In order to formally prove that diagnosis is able to strictly improve reliability we define the Dynamic Maximum Fault Assumption, which depends on the set of convicted nodes. We provide formal proofs in PVS that given the Dynamic Maximum Fault Assumption and a sane health status classification, the Interactive Consistency Protocol satisfies validity and agreement, and that the Diagnosis Protocol provides again a sane classification and convicts all benign faulty nodes, all accused symmetric faulty nodes, and asymmetric faulty nodes accused by a majority of undeclared nodes.

Keywords: fault tolerance, SPIDER, Byzantine, reliability, Diagnosis, Interactive Consistency.

1 Introduction

The Scalable Processor-Independent Design for Electromagnetic Resilience (SPIDER) is a family of general-purpose fault-tolerant architectures being designed at NASA Langley Research Center to support laboratory investigations into various recovery strategies from transient failures caused by electromagnetic effects. The core of the SPIDER architecture is the Reliable Optical Bus (ROBUS). As part of an effort partially sponsored by the FAA, the ROBUS is being developed in accordance with RTCA DO-254: Design Assurance Guidance for Airborne Electronic Hardware.

SPIDER is a family of communication architectures that provide reliable broadcast in the presence of multiple, possibly malicious ("Byzantine") faults (Figure 1). Various processing elements (PEs) are connected by the Reliable Optical Bus (ROBUS). The PEs may be computing nodes, sensors, or actuators, or composites of them. The ROBUS is formed by a column of N Bus Interface Units (BIUs), each connected to its PE, and a column of M Redundancy Management Units (RMUs). Each BIU is connected to each RMU, but the BIUs and RMUs are not connected to their own kind. In other words, the BIUs and the RMUs form a complete bipartite graph. The number of RMUs can be chosen freely; the sole purpose of the RMUs is fault tolerance.

SPIDER comes with three protocols: the Interactive Consistency Protocol, which takes care of reliable message broadcast, the Diagnosis Protocol, which arrives at a global fault classification, and the Synchronization Protocol, which synchronizes the clocks of all nodes. Synchronization provides a basis for us to compose nodes synchronously in a way similar to Rushby [4]. The SPIDER architecture and these three protocols

* This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the first author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681-2199, USA.

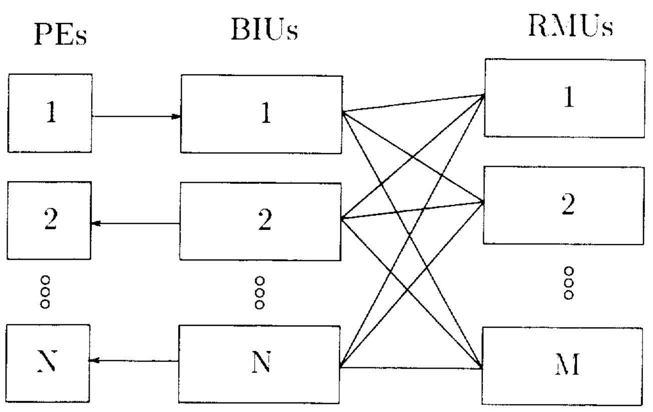


Fig. 1. SPIDER architecture

are described in more detail in [2]. Formal Verification of the first and the second protocol are complete: Formal Verification of the third is currently under development. A fourth protocol is being designed: the Readmission Protocol. Its purpose is to allow transiently faulty nodes to be reintegrated. All of the protocols are being verified using PVS [3].

In this paper we present two SPIDER protocols in detail: the Interactive Consistency Protocol and the Diagnosis Protocol. The PVS models for the two protocols can be found at URL

`http://www.icase.edu/~geser/spider/diag.dmp`

We state two essential assumptions. The first, called the Maximum Fault Assumption, ensures that the health status of the ROBUS is good enough for the protocols to work. The second, called the Correct Active Sources Assumption, takes care that the ROBUS has a sane view of its health status. We show that under these two assumptions,

- Interactive Consistency satisfies *validity* (the message of a good node is faithfully forwarded to all receivers) and *agreement* (all receivers receive the same message);
- Diagnosis preserves the Maximum Fault Assumption and the Correct Active Sources Assumption;
- Diagnosis provides convictions: every declared node is declared by all good nodes;
- Diagnosis declares all benign bad nodes, all accused symmetric bad nodes, and all asymmetric bad nodes accused by a good majority of undeclared nodes.

In the design of the two protocols we take care that readmission is not impeded. For instance, some good nodes may be distrusted for their former bad behavior. If there shall be a chance to readmit them, the distrust must not go on indefinitely. We address some of the ramifications. Moreover a readmission protocol is dynamic by nature, i.e., it has to speak about the temporal evolution of faults and their assessment. We do not elaborate on this issue here. Interactive Consistency and Diagnosis are modeled as functional programs. A dynamic model is outside the scope of this paper.

2 Related Work

The protocols for the SPIDER were derived from a number af earlier architectures. The Interactive Consistency protocol was inspired by the Draper FTP [6]. The initial PVS verification of the SPIDER protocol was adapted from the verification of the FTP protocol performed by Lincoln and Rushby [1]. The SPIDER diagnosis protocol was inspired by the on-line diagnosis protocol developed for MAFT [8]. Our diagnosis algorithm performs the same task as Algorithm DD in that paper. The PVS formalization of this protocol is described by Walter, Lincoln, and Suri [9]. For the verification of the MAFT architecture, gathering of accusations was explicitly modeled. We have separated the mechanism of fault detection from the distribution of the gathered fault information. Our notion of diagnosis refers to the latter. We give explicit constraints on what constitutes an accusation, i.e., the claim that a node is faulty. Any accusation basis that satisfies these constraints may be used.

The SPIDER architecture differs from many other fault-tolerant architectures in that it is not completely connected. So the BIUs have no direct observations of their own kind. This fact has fundamental ramifications. A BIU may only make direct accusations against RMUs. Any evidence that another BIU is faulty must come indirectly through the RMUs.

Rushby presents a comparison of four safety-critical bus architectures, including SPIDER, in [5].

3 Basic Types and Properties

3.1 Node Types and Symmetries

In the ROBUS architecture, BIUs and RMUs are somewhat symmetric to each other. This symmetry is not perfect since, e.g., BIUs are connected to PEs and RMUs are not. However it is tempting, and rewarding, to exploit in PVS the symmetries present.

The BIUs and the RMUs are instances of node types. Node types are modeled as types, *below*(K), of integers in the range $0..K-1$. On this account a node type is represented uniquely by its positive, finite cardinality, K. PVS theories may now be parameterized by node type cardinalities, which may be instantiated by N for the BIUs or by M for the RMUs. Sometimes we provide two parameters, L and R, for the node types LEFT and RIGHT, which may be instantiated by N and M, respectively, or by M and N, respectively. If the theory thus parameterized describes a protocol then the two instantiations mean usage of the protocol in two symmetric ways.

3.2 Hybrid Fault Types

Faults may be classified according to the potential consequences they may cause. Our approach is to assume that arbitrary, Byzantine faults may exist, but that they are rare and that less malicious faults come in greater numbers. We distinguish the following hybrid fault types, introduced in [7], that a node can exhibit:

- A **good** node behaves according to specification.
- A **benign** faulty node only sends messages that are detectably faulty (this includes nodes that have failed silent).
- A **symmetric** faulty node may send arbitrary messages, but does so the same way to each receiver.
- An **asymmetric** faulty node may send arbitrary messages that even may differ for the various receivers.

A node that is not good is called ***bad*** or ***faulty***. The three bad hybrid fault types form a hierarchy in the sense that an asymmetric node has all capabilities of a symmetric node, and a symmetric node may behave like a benign node.

In view of readmission, we further split good nodes into ***trustworthy*** nodes and ***recovering*** nodes. Both behave perfectly identical but the trustworthy nodes must moreover be trusted whereas the recovering nodes need not. The trustworthy nodes are the good ones that we can count on. Without readmission, all good nodes are trustworthy. With readmission, we may have a node that changes its fault status from *benign* to *good*. For various reasons SPIDER cannot figure out instantaneously that the node has changed. So there is some time interval where the node is already good but SPIDER has not yet noticed it. We let the node be *recovering* for some specified time to allow SPIDER to reset its trust in this node.

The sets of trustworthy, recovering, benign, symmetric, and asymmetric BIUs are denoted by TB, RB, BB, SB, AB, respectively. Likewise TR, RR, BR, SR, AR for the RMUs.

3.3 The Model of Communication

Data received by a node are of type *robus_data*$[T]$ where the parameter T denotes the type of data to be communicated. The type *robus_data*$[T]$ comprises

- **valid** data of type T,

- **receive error**, a token that expresses the fact that an error has been detected during reception,
- **source error**,
- **no majority**.

Note that the absence of an expected message can be detected, so a missing message is modeled by *receive_error*. The purpose of the tokens *source_error* and *no_majority* will be explained in the Interactive Consistency protocol below.

The behavior of a transmission line from LEFT node p to RIGHT node r, with data d to be sent, is modeled as a function *send* in PVS as follows:

```
$send(status)(p, d, r) : robus\_data[T] =$
    CASES $status(p)$ OF
        $trustworthy : d,$
        $recovering : d,$
        $benign : receive\_error,$
        $symmetric : sym\_send(p, d),$
        $asymmetric : asym\_send(p, d, r)$
    ENDCASES
```

status represents the global fault assignment to nodes, which during this discourse remains unchanged. We will henceforth suppress all status parameters in favor of a concise representation. As *send* must not use any information available to the hardware, we let *send* have no parameters that allow to deduce such information. Conversely, as we will see below, the hardware must not have any access to the fault status.

The value d is of type $robus\_data[T]$ rather than type T. This accounts for values that are relayed and may so be non-valid at p even before transmission. A benign node acts in such a way that a detectable error results at the receiver side. As we are not interested in the kind of error, we simply record the fact of the reception error. The behavior of a symmetric node is modeled by an uninterpreted function *sym_send*. By the definition of type $robus\_data[T]$, the value of $sym\_send(p, d)$ must be some of $valid(d')$, *receive_error*, *source_error*, *no_majority* for some $d' : T$. As we do not specify which it is, we have to cope with an arbitrary function *sym_send* which is constrained only by its parameters. The parameter d expresses the potential to send $valid(d)$, i.e., to fake good behavior. The parameter p expresses the potential that each symmetric node may exhibit individual behavior. The symmetric behavior follows from the fact that r is not a parameter of $sym\_send(p, d)$. For the asymmetric nodes we have a similar uninterpreted function *asym_send* which moreover has the parameter r and so the potential to exhibit asymmetric behavior.

3.4 How Faulty Node Behavior is Modeled

Bad behavior of a node is only observable through its communication. This gives us an extraordinary capability. We may rightfully pretend that a bad node works like a good node, and that all faulty behavior is due to the communication lines between nodes. Therefore we may speak about the state of a node as if it were a good node. And we may assume that each node sends correct values. Only upon reception does the bad character show up. We use this convenient view throughout the presentation.

3.5 Local Fault Classification

Each good node maintains a view of the health of all nodes. A node *obs* (the *observer*), may classify a node, *def* (the *defendant*), as being

- **trusted**, if *obs* has no evidence of *def* being faulty;
- **accused**, if *obs* knows that *def* is faulty but is uncertain whether this knowledge is shared by other good observers;
- **declared**, if *obs* knows that *def* is faulty and every good observer of the same kind also declares *def*.

The local fault classification forms a hierarchy of increasing knowledge of *obs* about *def*. Sometimes we will need to merge knowledge arriving from two sources; in this case we will get the maximum, w.r.t. the order *trusted* < *accused* < *declared*, of the arrived values. For instance, *merge*(*trusted*, *declared*) = *declared* and *merge*(*declared*, *accused*) = *declared*.

We call the map *act* that assigns each pair of nodes, *obs* and *def*, a fault classification, *act*(*obs*)(*def*), the *active-sources matrix* of the ROBUS. More precisely, there is an active sources matrix for each pair of node types of observer and defendant. The value *act*(*obs*)(*def*) is what *obs* takes *def* for in the current state. The row *act*(*obs*) of the active sources matrix is called the *active sources vector* of *obs*. Note that its active sources vector is all that *obs* knows about the health status of the ROBUS. The following three properties are required for active sources vectors, *actv*:

- **good trusting:** every good node is trusted.
- **symmetric agreement:** every non-asymmetric node is assessed the same way by two observers.[1]
- **declaration agreement:** the set of declared nodes is the same for any two observers.

The three conditions are rendered in PVS as follows:

$$\begin{aligned}
good\_trusting?(actv) : bool &= \forall def : good?(def) \Rightarrow trusted?(actv(def))\\
symmetric\_agreement?(actv_1, actv_2) : bool &= \forall def : \neg asymmetric?(def) \Rightarrow\\
&\qquad (trusted?(actv_1(def)) \Leftrightarrow trusted?(actv_2(def)))\\
declaration\_agreement?(actv_1, actv_2) : bool &= \forall def : declared?(actv_1(def)) \Leftrightarrow declared?(actv_2(def))
\end{aligned}$$

The conditions *good_trusting?* and *declaration_agreement?* are clearly motivated by our definitions of *accused?* and *declared?*. We will demonstrate in Example 2 that *symmetric_agreement?* is an essential premise to correctness of the Interactive Consistency Protocol.

In a new born ROBUS, no node has any evidence of faulty nodes whatsoever. Hence every node trusts every node. It is easy to verify that every row in this active sources matrix satisfies *good_trusting?*, and every pair of rows satisfies *symmetric_agreement?* and *declaration_agreement?*.

During the Diagnosis Protocol, active sources vectors are merged with new evidence, which is also represented as an active sources vector. The *merge* function is lifted to active sources vectors by

$$merge\_active(actv_1, actv_2) : active\_vector\_type[N] = \lambda def : merge(actv_1(def), actv_2(def)).$$

3.6 Message Qualities

For the proofs it will be convenient to speak about the following properties a message may have. A data vector may satisfy

- **good message:** a good node is assigned a valid message.
- **benign message:** a benign node is assigned *receive_error*.
- **benign source:** a benign node is assigned *source_error*.
- **good vote for:** all good nodes are assigned the same given message.

In PVS this is expressed by

$$\begin{aligned}
good\_message?(dv) : bool &= \forall G : good?(G) \Rightarrow valid?(dv(G))\\
benign\_message?(dv) : bool &= \forall G : benign?(G) \Rightarrow receive\_error?(dv(G))\\
benign\_source?(dv) : bool &= \forall G : benign?(G) \Rightarrow source\_error?(dv(G))\\
good\_vote\_for?(dv, d) : bool &= \forall G : good?(G) \Rightarrow dv(G) = d
\end{aligned}$$

[1] in view of the next property, it is sufficient to require that observers trust the same sets of non-asymmetric nodes.

3.7 Accusation Vectors

During the Diagnosis Protocol, messages are transmitted that carry information about the sender's classifications of other nodes. Usually only a certain aspect of the classification is wanted.

An accusation vector is a function from nodes to *accusations*, i.e., the values *working* or *failed*. Accusation vectors are sent across the ROBUS to inform other nodes about some aspects of the local active sources vector. To this end we provide two functions in PVS, one for encoding (*form*) and one for decoding (*extract*). For the purposes of accusation exchange, we encode the information whether a node is trusted. For the purposes of declaration exchange, we encode whether a node is declared:

$$\begin{aligned}
\mathit{form\_accvec}(\mathit{actv})(p) : \mathit{accusation} &= \text{IF } \mathit{trusted?}(\mathit{actv}(p)) \text{ THEN } \mathit{working} \\
&\quad\ \text{ELSIF } \mathit{accused?}(\mathit{actv}(p)) \text{ THEN } \mathit{failed} \\
&\quad\ \text{ELSE } \mathit{any\_accusation}(\mathit{actv}, p) \text{ ENDIF}, \\
\mathit{extract\_accvec}(\mathit{accv})(p) : \mathit{trust} &= \text{IF } \mathit{working?}(\mathit{accv}(p)) \text{ THEN } \mathit{trusted} \text{ ELSE } \mathit{accused} \text{ ENDIF}, \\
\mathit{form\_decvec}(\mathit{actv})(p) : \mathit{accusation} &= \text{IF } \mathit{declared?}(\mathit{actv}(p)) \text{ THEN } \mathit{failed} \text{ ELSE } \mathit{working} \text{ ENDIF}, \\
\mathit{extract\_decvec}(\mathit{accv})(p) : \mathit{trust} &= \text{IF } \mathit{working?}(\mathit{accv}(p)) \text{ THEN } \mathit{trusted} \text{ ELSE } \mathit{declared} \text{ ENDIF}
\end{aligned}$$

In the design of *form_accvec* we leave deliberately open how declarations are translated. This is achieved by the uninterpreted function *any_accusation*. In the current SPIDER implementation we took the choice $\mathit{any\_accusation}(\mathit{actv}, p) = \mathit{working}$. It is routine to prove a few lemmas that relate properties of accusation vectors to properties of active sources vectors.

3.8 The Maximum Fault Assumption

The purpose of the Maximum Fault Assumption is to both (1) hold with specified probability and (2) provide a basis to conclude correct operation of the SPIDER protocols. In the absence of diagnosis, the Maximum Fault Assumption is independent on the local knowledge of the health status of the system. SPIDER does not tolerate a simultaneous asymmetric fault of both a BIU and an RMU. Moreover we need a majority of trustworthy nodes of each kind. So the Maximum Fault Assumption is defined by:

1. $|TB| > |SB| + |AB|$, and
2. $|TR| > |SR| + |AR|$, and
3. $|AB| = 0$ or $|AR| = 0$.

This Maximum Fault Assumption does not improve by diagnosis, i.e., if diagnosis is able to declare new nodes then this has no impact on the Maximum Fault Assumption. Hence there is no formal proof that diagnosis has any positive effect. But there is a meta-level proof: assume that a node becomes benign when declared. Then declaring a symmetric or asymmetric node increases the majority of the good nodes, and declaring an asymmetric node may deplete the set of asymmetric nodes of its kind. This meta-level construction, however, does not allow readmission of transiently faulty nodes: a node that is diagnosed stays benign forever.

To enable readmission we introduce a weaker assumption that we call the Dynamic Maximum Fault Assumption. It has an additional parameter, U, for the set of nodes that some node does not declare. The Dynamic Maximum Fault Assumption is defined by:

1. $|TB| > |SB \cap U| + |AB \cap U|$, and
2. $|TR| > |SR \cap U| + |AR \cap U|$, and
3. $|AB \cap U| = 0$ or $|AR \cap U| = 0$.

Note that diagnosis improves the Dynamic Maximum Fault Assumption. As more nodes are declared, U gets smaller, and so it is easier to have a majority of good nodes among U, or to exclude asymmetric nodes from U. Our hope is that diagnosis will particularly aid in the latter.

4 An Informal Overview

4.1 The Interactive Consistency Protocol

A message broadcast protocol is called reliable if it satisfies the following two properties:

- **validity:** every good node receives the value sent by a good node.
- **agreement:** all good nodes agree in the value sent.

For reliable message passing in the presence of various faults, we use the Interactive Consistency Protocol. We present the protocol parameterized by two node types, LEFT and RIGHT, that may be instantiated by the types of BIUs and RMUs, respectively, or by the types of RMUs and BIUs, respectively. For reliable message broadcast among the PEs, we need the former instance. Indeed, in the Diagnosis Protocol we will need both symmetric instances. The Interactive Consistency Protocol works as follows:

1. A single LEFT node, g, as per agreed schedule, broadcasts some value, *valid*(v), to all RIGHT nodes.
2. Every RIGHT node relays its received value, d, to all LEFT nodes. However, if $d =$ *receive_error* then it sends *source_error* to redirect the blame from itself to the originator of the message.
3. Every LEFT node, p, collects the vector of values it received (one value per RIGHT node). Then it determines the set of RIGHT nodes it trusts. A RIGHT node from which p receives *receive_error* is accused by p. These classifications are merged with p's Active Sources Vector.
4. Else if p receives some value d_{maj} from a majority of trusted RIGHT nodes then it determines d_{maj} and, if d_{maj} is non-valid, declares g. Otherwise p determines *no_majority* and declares g.
5. If LEFT g is declared by p (not including the recent declarations in Step 4) then p forwards *source_error* to its PE. Otherwise p forwards the value determined in Step 4 to its PE.

We will refer to Step 1 as a *single source broadcast*, and to Steps 2 to 4 as a *consistent source exchange*. Consistent source exchange will turn out useful outside the Interactive Consistency Protocol.

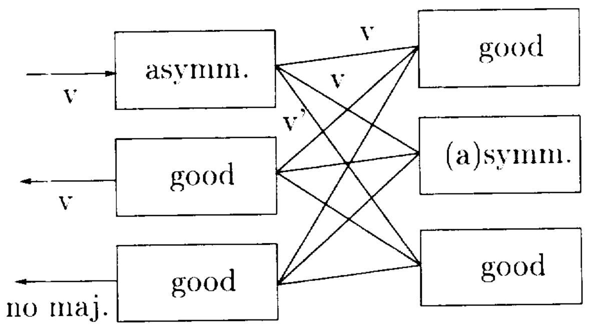


Fig. 2. Two counterexamples: Step 5 is essential for agreement (RMU 2 is asymmetric; see Example 1) and the Symmetric Agreement premise is essential for agreement (RMU 2 is symmetric; see Example 2)

In a previous version of the Interactive Consistency Protocol, in Step 2 a relay turned d into *source_error* also in the case where it received a correct value but did not trust the general G. Although this is undeniably correct, it is undesirable in view of readmission because a declared node has no chance to prove it is good again.

Step 5 is introduced to ensure agreement under the weaker Dynamic Maximum Fault Assumption. A version that skips Step 5 and rather forwards the value determined in Step 4 to the PEs, may violate agreement under the Dynamic Maximum Fault Assumption:

Example 1. Let $M = N = 3$ (Figure 2). Suppose that BIU 1 is asymmetric, but declared; the Dynamic Maximum Fault Assumption allows the existence of asymmetric, undeclared RMUs. Now let BIU 1 send *valid*(v) to the good RMU 1 and to the asymmetric RMU 2, and a different *valid*(v') to the good RMU 3.

Let BIU 2 trust all RMUs, and let BIU 3 trust RMUs 1 and 3 only. Then BIU 2 finds a majority value, *valid*(v), whereas BIU 3 finds no majority. So the old Interactive Consistency Protocol does not satisfy agreement for the Dynamic Maximum Fault Assumption.

A similar example shows that the *symmetric_agreement?* premise is essential:

Example 2. Let $M = N = 3$. Suppose that BIU 1 is asymmetric and the static Maximum Fault Assumption holds. Now let BIU 1 send *valid*(v) to the good RMU 1 and to the symmetric RMU 2, and a different *valid*(v') to the good RMU 3. Let BIU 2 trust all RMUs, and let BIU 3 trust RMUs 1 and 3 only. So symmetric agreement does not hold. Then BIU 2 finds a majority value, *valid*(v), whereas BIU 3 finds no majority. So agreement is violated.

4.2 On the Collection of Evidence

During all protocols each node records evidence of faulty behavior of other nodes that it learns through communication. Some of this evidence may lead to an accusation of a node. Some evidence, for instance a non-valid result of an Interactive Consistency exchange, may even lead to a *declaration* of the general by virtue of the agreement property.

We distinguish between direct and indirect observations that lead to accusations against nodes. A *direct observation* is a single event that may lead to the accusation of the sender. Direct observations are:

- No message was received during the reception window;
- An improperly formatted message is received.

These observations take place during all protocols. "Improperly formatted" may also mean that an encoded message does not pass the parity check. The effect of a direct observation is modeled by the *receive_error* token.

Indirect observations are a collection of events that together provide the basis of an accusation. This involves a majority vote. They come in two kinds: either a node of the same kind, or a node of the other kind is accused. Let the diagnosing node be a LEFT node.

1. a majority of RIGHT nodes offer evidence against a LEFT node; that LEFT node is accused.
2. in a consistent source exchange, a RIGHT node disagrees with the majority; this node is accused.
3. in an Interactive Consistency exchange from the LEFT, there is no majority; then the general is accused.
4. in a majority of Interactive Consistency exchanges from the LEFT, a RIGHT relay disagrees with the majority; this relay is accused.

By the agreement property, an accusation of the Form 3 is made by all LEFT observers, so they may issue a declaration. We conjecture that, likewise, an accusation of Form 1 can be turned into a declaration. A typical case is the declaration of a general of an Interactive Consistency exchange on the basis that a majority of trusted RIGHT nodes vote for *source_error*. All of these indirect observation mechanisms are in place in the current SPIDER implementation.

4.3 The Diagnosis Protocol

The purpose of the SPIDER Diagnosis Protocol is to distribute the local accusations in order to arrive at a consistent classification of the health status of the ROBUS. It does that by merging the active sources vectors with new declarations. By the bipartite ROBUS architecture, agreement among all nodes cannot be reached in one sweep. Rather, agreement among nodes of the same kind is achieved first, i.e., we get declarations. This step is called the *accusation exchange.* Agreement among nodes of opposite kind is achieved in the second sweep, the *declaration exchange.*

The Diagnosis Protocol shall preserve *good_trusting?*, *symmetric_agreement?*, and *declaration_agreement?*. By the fact that classifications are merged, the Maximum Fault Assumption is also preserved. We require moreover that the Diagnosis Protocol establishes the following properties:

- **conviction agreement:** every declared node is convicted, i.e., it is declared by both kinds of nodes.

- **benign declaring:** every benign node is declared.
- **symmetric declaring:** every symmetric node that is accused by a good node is declared.
- **majority declaring:** if a set of good LEFT nodes that forms a majority among the undeclared and non-benign LEFT nodes accuses a node then that node is declared. Likewise for RIGHT nodes.

The properties are rendered in PVS as follows:

$$\begin{aligned}
&\mathit{conviction\_agreement?}(\mathit{act}_{ll}, \mathit{act}_{lr}) : \mathit{bool} = \\
&\quad \forall q, r : \mathit{good?}(q)\ \&\ \mathit{good?}(r) \Rightarrow \mathit{declaration\_agreement?}(\mathit{act}_{ll}(q), \mathit{act}_{lr}(r)). \\
&\mathit{benign\_declaring?}(\mathit{actv}_d) : \mathit{bool} = \forall \mathit{def} : \mathit{benign?}(\mathit{def}) \Rightarrow \mathit{declared?}(\mathit{actv}(\mathit{def})) \\
&\mathit{symmetric\_declaring?}(\mathit{actv}_{d1}, \mathit{actv}_{d2}) : \mathit{bool} = \\
&\quad \forall \mathit{def} : \mathit{accused?}(\mathit{actv}_{d1}(\mathit{def}))\ \&\ \mathit{symmetric?}(\mathit{def}) \Rightarrow \mathit{declared?}(\mathit{actv}_{d2}(\mathit{def})) \\
&\mathit{majority\_declaring?}(\mathit{actv}_{l1}, \mathit{actv}_{d1}, \mathit{actv}_{d2}) : \mathit{bool} = \\
&\quad \forall \mathit{def} : \neg \mathit{working?}(\mathit{majority}(\mathit{undeclared}(\mathit{actv}_{l1}) \setminus \mathit{benign}, \mathit{working}, \\
&\qquad\qquad\qquad\qquad \mathit{solid}(\mathit{form\_accvec}(\lambda G : \mathit{actv}_{dl}(G)(\mathit{def}))))) \\
&\qquad \Rightarrow \mathit{declared?}(\mathit{act}_{d2}(\mathit{def}))
\end{aligned}$$

where the auxiliary function *solid* is defined by

$$\mathit{solid}(\mathit{accv})(G) : \mathit{accusation} = \text{IF } \mathit{good?}(G) \text{ THEN } \mathit{accv}(G) \text{ ELSE } \mathit{working} \text{ ENDIF}$$

The operator \ denotes set difference.

The requirements are motivated as follows. *conviction_agreement?* expresses the maximal distribution of declarations: every node arrives at the same set. Of course we want to declare as many bad nodes as possible: all benign nodes, all symmetric nodes accused by a good node, and some asymmetric nodes. Example 3 below shows that in the property *majority_declaring?*, it is essential that the accusing nodes that are in the majority are good. In other words, it is essential that the accusing nodes that form a majority are good. *solid(accv)* denotes the set of accusations in *accv* that cannot be shattered.

The Diagnosis Protocol works as follows:

1. BIUs reliably exchange their accusations. If a majority of undeclared BIUs accuse a node then that node is declared.
2. RMUs reliably exchange their accusations. If a majority of undeclared RMUs accuse a node then that node is declared.
3. BIUs broadcast their declarations to the RMUs; the RMUs merge these with their declarations.
4. RMUs broadcast their declarations to the BIUs; the BIUs merge these with their declarations.

We exploit the obvious symmetry by providing an *accusation exchange* protocol and a *declaration exchange* protocol, each parameterized with LEFT and RIGHT.

The accusation exchange protocol works as follows:

1. each LEFT node, G, uses the Interactive Consistency Protocol to broadcast, to all LEFTs, its vector of accusations against any node.
2. after Step 1, each LEFT node, p, has collected a vector of received values (one value per general G).
3. LEFT nodes from which p received *source_error* are declared by p.
4. If a majority of (then) undeclared LEFT nodes accuse a node then that node is declared.

The declaration exchange protocol works as follows:

1. by a consistent source exchange, each RIGHT node, r, broadcasts its declaration vector to all LEFT nodes.
2. each LEFT node, p, merges the received declarations with its own active sources vector.

The following counterexample shows that a majority of undeclared accusers may or may not lead to the declaration of a node. If a bad accuser is tossed out of the set of undeclared nodes then the majority may get lost. Even worse, the opposite of its vote may be received. In other words, for property *majority_declaring?* it must be good nodes that form the majority.

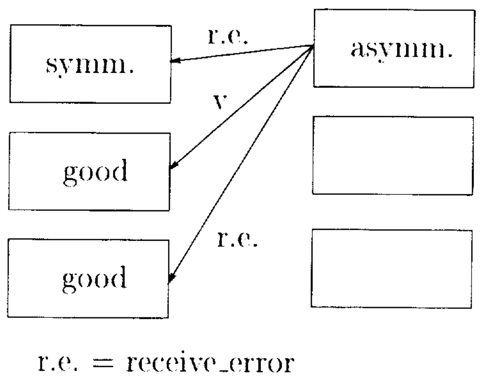


Fig. 3. A majority that is supported by a symmetric node (see Example 3)

Example 3. Let $M = N = 3$ (Figure 3). Suppose that BIU 1 is symmetric, that BIUs 2 and 3 are good, and that BIUs have no declarations against each other. Next suppose that RMU 1 is asymmetric and trusted by the BIUs. We do not care about the other RMUs. Now let the BIUs receive *receive_error*, *valid*(v), and *receive_error*, respectively, from RMU 1 during some Interactive Consistency exchange. This has the BIUs 2 and 3 accuse RMU 1. We now show that RMU 1 is declared if and only if BIU 1 is not excluded from the vote of the BIUs. If the RMUs receive *receive_error* from BIU 1, then they forward *source_error* to all BIUs, so all good BIUs declare BIU 1. But then the number of votes (one) that accuse RMU 1 is not sufficient to form a majority in the set {BIU 2, BIU 3}. So RMU 1 is not declared. However if the RMUs receive a correct accusation vector from BIU 1 then the number of votes (two) is sufficient for a majority in the set of all BIUs, and RMU 1 is declared.

Example 3 witnesses the strange fact that diagnosis may profit from a faulty node (BIU 1) going undetected.

5 The Formal Model

The following description is intended to aid the reader in understanding the PVS proofs.

5.1 Majority Votes

Suppose that every LEFT node presents a value to our receiving node. These values form a value vector, vv. The set of voters for a value v is defined by $\{i \mid vv(i) = v\}$. Some voters can be excluded from the vote by a filter set H; only votes of members of H are counted. Now v is in the majority among H in vv, if $2|H \cap \{i \mid vv(i) = v\}| > |H|$. The function $\mathit{majority}(H, \mathit{default}, vv)$ yields this unique majority value v if it exists, otherwise it yields the default value, $\mathit{default}$. Two value vectors are called *H-similar* if they agree for nodes that are in H. Formally:

$$\mathit{similar\_vector?}(H, vv_1, vv_2) : \mathit{bool} = \forall i : H(i) \Rightarrow vv_1(i) = vv_2(i)$$

The outcome of the vote agrees for similar vectors:

Lemma 1 (Majority unique).

$$\mathit{similar\_vector?}(H, vv_1, vv_2) \vdash \mathit{majority}(H, \mathit{default}, vv_1) = \mathit{majority}(H, \mathit{default}, vv_2)$$

If the good nodes (of one kind) agree, then the value they vote will be the majority value:

Lemma 2 (Good vote for majority).

$$\begin{array}{l} \quad \mathit{hybrid\_majority\_good?}(H), \mathit{hybrid\_select?}(H), \\ \quad \mathit{good\_vote\_for?}(vv, v) \\ \vdash \\ \quad \mathit{majority}(H, \mathit{default}, vv) = v \end{array}$$

There is one variant of Lemma 2 per node kind. For the BIUs, *hybrid_majority_good?*(H) stands for $|TB| > |SB \cap H| + |AB \cap H|$; *hybrid_select?*$(H)$ denotes that $TB \subset H$ and $BB \cap H = \emptyset$. The two properties together ensure that $2|TB \cap H| > |H|$, i.e., the good are in the majority among H. Since the set of good nodes is a subset of the set of nodes that vote for v, the value v will be the majority value.

5.2 Consistent Source Exchange

In view of their role in the Interactive Consistency Protocol, we call the transmitters in a consistent source exchange *relays*.

Suppose that dv denotes the vector of values sent by the relays. The function *relay_data*, defined by

$$\mathit{relay\_data}(dv)(r)(G) : \mathit{robus\_data}[T] = \mathit{input\_filter}(\mathit{send}(G, dv(G), r)),$$

describes the values received at r and preprocessed by *input_filter*. The preprocessing replaces all unexpected message formats, such as *no_majority* or *source_error*, by *receive_error*. The function *elim_relays*$(dv, actv_l)$ merges $actv_l$, the active sources vector of the receiver, with the fresh evidence obtained from the received data vector dv: a node from which *receive_error* was received is accused. The function *vote*, defined by

$$\mathit{vote}(dv, actv_l) : \mathit{robus\_data}[T] = \mathit{majority}(\mathit{trusted}(\mathit{elim\_relays}(dv, actv_l)), \mathit{no\_majority}, dv),$$

yields the majority value among the trusted nodes if it exists, and *no_majority* else. The consistent source exchange is modeled in PVS as a function

$$\mathit{csx}(dv, actv_l)(r) : \mathit{robus\_data}[T] = \mathit{vote}(\mathit{relay\_data}(dv)(r), actv_l)$$

The following theorem assumes that a RIGHT node, r, trusts all good nodes; every good LEFT node G sends a valid message $dv(G)$; all good nodes send the same value d; the set of trusted nodes has greater cardinality than the set of symmetric or asymmetric nodes. It says that then r determines the value d. The active sources vector of r is denoted by $actv_l$. The vector of data that the LEFT nodes send is given by dv.

We will abbreviate *elim_relays*$(\mathit{relay\_data}(dv)(r), actv_l)$ by *elim_csx_relays*$(dv, actv_l)(r)$.

Theorem 1 (Validity of consistent source exchange).

$$\begin{array}{l} \quad \mathit{good\_trusting?}(actv_l), \mathit{good\_message?}(dv), \mathit{good\_vote\_for?}(dv, d), \\ \quad \mathit{hybrid\_majority\_good?}(\mathit{trusted}(\mathit{elim\_csx\_relays}(dv, actv_l)(r))), \\ \vdash \\ \quad \mathit{csx}(dv, actv_l)(r) = d \end{array}$$

Proof. relay_data has the property *benign_message?* and preserves the pair of properties *good_message?* and *good_vote_for?*. Next *trusted*$(\mathit{elim\_csx\_relays}(dv, actv_l)(r))$ satisfies *hybrid_select?*. Lemma 2 finishes the proof.

The following lemma is crucial for the agreement theorem below. It states that if two right nodes trust no asymmetric left node; trust the same symmetric left nodes; and receive the same data vectors, up to data from non-trusted relays, then they will finally trust the same left nodes.

Lemma 3 (Trusted agreement).

$$\begin{array}{ll} & \mathit{no\_asymmetric?}(\mathit{trusted}(\mathit{actv}_{l1})), \mathit{no\_asymmetric?}(\mathit{trusted}(\mathit{actv}_{l2})), \qquad (1)\\ & \mathit{symmetric\_agreement?}(\mathit{actv}_{l1}, \mathit{actv}_{l2}), \mathit{similar\_vector?}(\mathit{trusted}(\mathit{actv}_{l1}), \mathit{dv}_1, \mathit{dv}_2),\\ \vdash & \\ & \mathit{trusted}(\mathit{elim\_relays}(\mathit{dv}_1, \mathit{actv}_{l1})) = \mathit{trusted}(\mathit{elim\_relays}(\mathit{dv}_2, \mathit{actv}_{l2})) \end{array}$$

Proof. Let x be a node. We prove that x is trusted by the active sources vector $\mathit{elim\_relays}(\mathit{dv}_1, \mathit{actv}_{l1})$ iff x is trusted by $\mathit{elim\_relays}(\mathit{dv}_1, \mathit{actv}_{l2})$. If x is asymmetric then by (1), x is trusted neither by actv_{l1} nor by actv_{l2}. Because $\mathit{elim\_relays}(\mathit{dv}_1, \mathit{actv}_l)$ trusts less nodes than actv_l for all actv_l, we get the claim. So let x be non-asymmetric. Then by symmetric agreement, $\mathit{actv}_{l1}(x) = \mathit{actv}_{l2}(x)$. If actv_{l1} trusts x, then $\mathit{dv}_1(x) = \mathit{dv}_2(x)$ because $\mathit{dv}_1, \mathit{dv}_2$ are $\mathit{trusted}(\mathit{act}_{l1})$-similar. It turns out that then the relays disqualified by $\mathit{dv}_1, \mathit{dv}_2$ are the same. Finally, if actv_{l1} does not trust x, neither $\mathit{elim\_relays}(\mathit{dv}_1, \mathit{actv}_{l1})$ nor $\mathit{elim\_relays}(\mathit{dv}_1, \mathit{actv}_{l2})$ trusts x.

Theorem 2 (Agreement of consistent source exchange).

$$\begin{array}{ll} & \mathit{good\_trusting?}(\mathit{actv}_{l1}), \mathit{good\_trusting?}(\mathit{actv}_{l2}),\\ & \mathit{symmetric\_agreement?}(\mathit{actv}_{l1}, \mathit{actv}_{l2}), \mathit{good\_message?}(\mathit{dv}),\\ & \mathit{no\_asymmetric?}(\mathit{trusted}(\mathit{actv}_{l1}))\ \&\ \mathit{no\_asymmetric?}(\mathit{trusted}(\mathit{actv}_{l2}))\ \mathrm{OR} \qquad (2)\\ & \quad \mathit{good\_vote\_for?}(\mathit{dv})\ \&\ \mathit{hybrid\_majority\_good?}(\mathit{trusted}(\mathit{elim\_csx\_relays}(\mathit{dv}, \mathit{actv}_{l1})(r_1)))\ \& \qquad (3)\\ & \quad \mathit{hybrid\_majority\_good?}(\mathit{trusted}(\mathit{elim\_csx\_relays}(\mathit{dv}, \mathit{actv}_{l2})(r_2))), \qquad (4)\\ \vdash & \\ & \mathit{csx}(\mathit{dv}, \mathit{actv}_{l1})(r_1) = \mathit{csx}(\mathit{dv}, \mathit{actv}_{l2})(r_2) \end{array}$$

Proof. By case analysis. If (2) then the vectors received by any two RIGHT nodes are similar. Lemma 3 establishes that the sets of trusted nodes agree for r_1 and r_2. Hence r_1 and r_2 will arrive at the same result of the voting. If rather (3) and (4) hold, then $\mathit{csx}(\mathit{dv}, \mathit{actv}_{l1})(r_1) = \mathit{dv}(p) = \mathit{csx}(\mathit{dv}, \mathit{actv}_{l2})(r_2)$ for some good LEFT p by two applications of Theorem 1.

For curiosity we mention the remarkable fact that only the most up-to-date set of trusted nodes, $\mathit{elim\_csx\_relays}(\mathit{dv}, \mathit{actv}_l)(r)$, needs to satisfy the $\mathit{hybrid\_majority\_good?}$ property. This is the best behavior one may ask for. However in favor of a simple approach we will replace this premise to the more conservative $\mathit{hybrid\_majority\_good?}(\mathit{trusted}(\mathit{actv}_l)))$.

5.3 Single Source Broadcast and Interactive Consistency

Single source broadcast is modeled in PVS as a function

$$\mathit{source\_data}(d, G)(r) : \mathit{robus\_data}[T] = \mathit{source\_filter}(\mathit{input\_filter}(\mathit{send}(G, d, r)))$$

where $\mathit{source\_filter}$ replaces $\mathit{receive\_error}$ by $\mathit{source\_error}$ and leaves all other messages unchanged. We will refer to the sender, G, of the broadcast as the *general.*

The Interactive Consistency Protocol is then summarized in the function

$$\begin{array}{l} \mathit{hom}(\mathit{act}, \mathit{actv}_r, G, v, p) : \mathit{robus\_data}[T] =\\ \quad \mathrm{IF}\ \mathit{declared?}(\mathit{act})\ \mathrm{THEN}\ \mathit{source\_error}\ \mathrm{ELSE}\ \mathit{csx}(\mathit{source\_data}(\mathit{valid}(v), G), \mathit{actv}_r)(p)\ \mathrm{ENDIF} \end{array}$$

We will abbreviate $\mathit{elim\_csx\_relays}(\mathit{source\_data}(\mathit{valid}(v), G), \mathit{actv}_r, p)$ by $\mathit{elim\_ic\_relays}(\mathit{actv}_r, G, v, p)$. Interactive Consistency satisfies Validity and Agreement:

Theorem 3 (Validity of Interactive Consistency).

$$\begin{array}{l} \mathit{hybrid\_majority\_good?}(\mathit{trusted}(\mathit{elim\_ic\_relays}(\mathit{actv}_r, G, v, p))), \\ \mathit{good\_trusting?}(\mathit{actv}_r), \\ \neg \mathit{asymmetric}(G), \neg \mathit{declared?}(\mathit{act}) \\ \vdash \\ \mathit{hom}(\mathit{act}, \mathit{actv}_r, G, v, p) = \mathit{source\_data}(\mathit{valid}(v), G) \end{array}$$

Proof. Each RIGHT node will get the same message, $\mathit{source\_data}(\mathit{valid}(v), G, r)$, since the general, G, is not asymmetric. So all good RIGHT nodes vote for this message. Moreover *source_data* satisfies *benign_source?* and preserves *good_message?*. The claim follows by Theorem 1.

Theorem 4 (Agreement of Interactive Consistency).

$$\begin{array}{l} \mathit{no\_asymmetric?}(\mathit{trusted}(\mathit{actv}_{r1}))\ \&\ \mathit{no\_asymmetric?}(\mathit{trusted}(\mathit{actv}_{r2}))\ \mathrm{OR} \\ \quad \mathit{declared?}(\mathit{act}_1)\ \mathrm{OR} \\ \quad \neg \mathit{asymmetric}(G)\ \& \\ \quad \mathit{hybrid\_majority\_good?}(\mathit{trusted}(\mathit{elim\_ic\_relays}(\mathit{actv}_{r1}, G, v, p_1)))\ \& \\ \quad \mathit{hybrid\_majority\_good?}(\mathit{trusted}(\mathit{elim\_ic\_relays}(\mathit{actv}_{r2}, G, v, p_2))), \\ \mathit{good\_trusting?}(\mathit{actv}_{r1}), \mathit{good\_trusting?}(\mathit{actv}_{r2}), \\ \mathit{symmetric\_agreement?}(\mathit{actv}_{r1}, \mathit{actv}_{r2}), \\ \mathit{declared?}(\mathit{act}_1) = \mathit{declared?}(\mathit{act}_2) \\ \vdash \\ \mathit{hom}(\mathit{act}_1, \mathit{actv}_{r1}, G, v, p_1) = \mathit{hom}(\mathit{act}_2, \mathit{actv}_{r2}, G, v, p_2) \end{array}$$

Proof. If the general, G, is declared by p_1 or by p_2 then it is declared by each. So $\mathit{hom}(\mathit{act}_1, \mathit{actv}_{r1}, G, v, p_1) = \mathit{source\_error} = \mathit{hom}(\mathit{act}_2, \mathit{actv}_{r2}, G, v, p_2)$.

If G is not asymmetric, each RIGHT node will get the same message, $\mathit{source\_data}(\mathit{valid}(v), G, r)$. So all good RIGHT nodes vote for this message. Moreover *source_data* satisfies *benign_source?* and preserves *good_message?*. By Theorem 2 the claim follows.

As corollaries from Theorem 3 we get that the value sent by good nodes is indeed forwarded by all LEFTs to their PEs; *source_error* is forwarded if the general is benign: and the same value, $\mathit{sym\_send}(g, \mathit{valid}(v))$, is forwarded if the general is undeclared and symmetric.

5.4 Accusation Exchange

Accusation Exchange is split into two parts: the reliable passing of accusation messages, and their voting.

The following PVS function provides accusation message passing:

$$\begin{array}{l} \mathit{mxfer}(\mathit{act}_{ll}, \mathit{act}_{rl}, \mathit{act}_{dl})(p)(G) : \mathit{robus\_data}[\mathit{accusation\_vector\_type}[D]] = \\ \qquad \mathit{hom}(\mathit{act}_{ll}(p)(G), \mathit{act}_{rl}(p))(G, \mathit{form\_accvec}(\mathit{act}_{dl}(G)), p) \end{array}$$

Here every general, G, uses the Interactive Consistency Protocol to communicate its accusation vector to every receiver, p. The accusation vector is formed from G's active sources vector, $\mathit{act}_{dl}(G)$. The receiver node uses its active sources vector, $\mathit{act}_{rl}(p)$, to identify the trusted relays, and the trust value $\mathit{act}_{ll}(p)(G)$, to decide whether G is declared.

The received accusation vectors are then processed as follows. First the received vector of messages, one message per LEFT node, is unpacked by *unpack_vec*, defined by

$$\begin{array}{l} \mathit{unpack\_vec}(v) : \mathit{accusation\_vector\_type}[D] = \\ \quad \mathrm{IF}\ \mathit{valid}(v)\ \mathrm{THEN}\ \mathit{value}(v)\ \mathrm{ELSE}\ \mathit{any\_accusation}(v)\ \mathrm{ENDIF} \end{array}$$

If v is a valid message then its value, an accusation vector, can be extracted faithfully. Else we return an arbitrary value by the uninterpreted function $\mathit{any\_accusation}(v) : \mathit{accusation\_vector\_type}[D]$. A node can declare a source from which it receives a non-valid message:

$$\begin{aligned}&\mathit{disqualified\_sources}(dv) : \mathit{active\_vector\_type}[L] = \\ &\quad \lambda G : \text{IF } \mathit{valid}(dv(G)) \text{ THEN } \mathit{trusted} \text{ ELSE } \mathit{declared} \text{ ENDIF}\end{aligned}$$

This declaration set is merged with the active sources vector of the receiver:

$$\begin{aligned}&\mathit{elim\_disqualified}(dv, \mathit{actv}_l) : \mathit{active\_vector\_type}[L] = \\ &\quad \mathit{merge\_active}(\mathit{disqualified\_sources}(dv), \mathit{actv}_l)\end{aligned}$$

The accusation vectors obtained from the undeclared sources form what we call the receiver's *accusation matrix*. A row being the accusation vector received from a node (the *accuser*), the columns form the vectors of accusations against a node (the *defendant*). For each defendant, def, the receiver does a majority vote on the column of the accusation matrix. Only the votes of undeclared nodes are counted.

$$\begin{aligned}&\mathit{court}(\mathit{actv}_l, dv) : \mathit{accusation\_vector\_type}[D] = \\ &\quad \lambda \mathit{def} : \mathit{majority}(\mathit{undeclared}(\mathit{actv}_l), \mathit{working})(\lambda G : \mathit{unpack\_vec}(dv(G))(\mathit{def}))\end{aligned}$$

The default value of the majority vote is $\mathit{working}$, in order to implement the principle "not guilty until proven guilty". The full voting complex of the accusation exchange is modeled as

$$\begin{aligned}&\mathit{accx}(dv, \mathit{actv}_l, \mathit{actv}_d) : \mathit{active\_vector\_type}[D] = \\ &\quad \mathit{merge\_active}(\mathit{extract\_decvec}(\mathit{court}(\mathit{elim\_disqualified}(dv, \mathit{actv}_l), dv)), \mathit{actv}_d)\end{aligned}$$

The complete accusation exchange is the composition of the transmission with the processing primitive:

$$\begin{aligned}&\mathit{accx\_combo}(\mathit{act}_{ll}, \mathit{act}_{rl}, \mathit{act}_{dl})(p) : \mathit{active\_vector\_type}[D] = \\ &\quad \mathit{accx}(\mathit{mxfer}(\mathit{act}_{ll}, \mathit{act}_{rl}, \mathit{act}_{dl})(p), \mathit{act}_{ll}(p), \mathit{act}_{dl}(p))\end{aligned}$$

Under reasonable assumptions, accusation exchange satisfies $\mathit{good\_trusting?}$, $\mathit{symmetric\_agreement?}$, and $\mathit{declaration\_agreement?}$. To reduce technical clutter it is useful to split the proofs into lemmas about the components, accx and mxfer. We render here only the most involved proof of the three:

Theorem 5.

$$\begin{aligned}&\mathit{hybrid\_majority\_good?}(\mathit{undeclared}(\mathit{elim\_disqualified}(dv, \mathit{actv}_l))), \\ &\mathit{good\_trusting?}(\mathit{actv}_d), \mathit{good\_trusting?}(\mathit{actv}_l), \\ &\mathit{good\_message?}(dv), \mathit{benign\_source?}(dv), \mathit{accusation\_message?}(dv) \\ \vdash \\ &\mathit{good\_trusting?}(\mathit{accx}(dv, \mathit{actv}_l, \mathit{actv}_d))\end{aligned}$$

Proof. From the premises $\mathit{good\_message?}(dv)$, $\mathit{benign\_source?}(dv)$, and $\mathit{good\_trusting?}(\mathit{actv}_l)$, the property $\mathit{hybrid\_select?}(\mathit{undeclared}(\mathit{elim\_disqualified}(dv, \mathit{actv}_l)))$ follows. By the definitions of $\mathit{merge\_active}$ and of $\mathit{extract\_decvec}$, we have to prove that court associates $\mathit{working}$ to each good defendant, def. So let def be a good node. We have to show that no majority of undeclared sources accuses def. By the premise $\mathit{accusation\_message?}(dv)$, all good LEFT nodes agree in that def is $\mathit{working}$. By Lemma 2 the claim follows.

Given $\mathit{hybrid\_majority\_good?}(\mathit{trusted}(\mathit{act}_{rl}(p)))$, $\mathit{good\_trusting?}(\mathit{act}_{ll}(p))$, $\mathit{good\_trusting?}(\mathit{act}_{rl}(p))$, and $\mathit{good?}(G)$, the result of mxfer satisfies $\mathit{good\_message?}$, $\mathit{benign\_source?}$, and $\mathit{accusation\_message?}$. So we may conclude:

Theorem 6.

$$\begin{aligned}
&\mathit{hybrid\_majority\_good?}(\mathit{undeclared}(\mathit{elim\_disqualified}(dv, act_{ll}(p)))),\\
&\mathit{hybrid\_majority\_good?}(\mathit{trusted}(act_{rl}(p))),\\
&\mathit{good\_trusting?}(act_{dl}(p)),\ \mathit{good\_trusting?}(act_{ll}(p))\\
\vdash\\
&\mathit{good\_trusting?}(\mathit{accx\_combo}(act_{ll}, act_{rl}, actv_{dl})(p))
\end{aligned}$$

5.5 Like Accusation Exchange

The exchange of accusation against unlike nodes is exactly what we have just sketched. Exchange of accusation against like nodes has the additional Step 3. We model it by a merge of the accusation exchange with the new evidence.

$$\begin{aligned}
&\mathit{accx\_combo\_like}(act_{ll}, act_{rl})(p) : \mathit{active\_vector\_type}[L] =\\
&\quad \mathit{merge\_active}(\mathit{merge\_active}(\mathit{disq\_combo}(act_{ll}, act_{rl}, act_{ll})(p),\\
&\qquad\qquad\qquad\qquad \mathit{disq\_combo}(act_{ll}, act_{rl}, act_{rl})(p)),\\
&\qquad\quad \mathit{accx\_combo}(act_{ll}, act_{rl}, act_{ll})(p))
\end{aligned}$$

The active sources vector $\mathit{disq\_combo}(act_{ll}, act_{rl}, act_{ll})(p)$ expresses the declarations uttered against the generals from which *source_error* was received during the accusation exchange against LEFT (i.e., like) nodes. Similar declarations can be made during the accusation exchange against RIGHT nodes, yielding $\mathit{disq\_combo}(act_{ll}, act_{rl}, act_{rl})(p)$. The two active sources vectors are merged to the *accx_combo* result.

It is straightforward to show that this merge preserves all properties claimed for *accx_combo*. So these properties also hold for *accx_combo_like*.

5.6 Declaration Exchange

The reliable transfer of the declarations to the opposite kind of nodes is done by a consistent source exchange. Each node, G, sends its vector of declarations against defendant nodes to all unlike nodes, r.

$$\begin{aligned}
&\mathit{declxfer}(act_{dl}, act_{lr})(r) : \mathit{robus\_data}[\mathit{accusation\_vector\_type}[D]] =\\
&\quad \mathit{csx}(\lambda G : \mathit{valid}(\mathit{form\_decvec}(act_{dl}(G))), act_{lr}(r))(r)
\end{aligned}$$

Theorems 1 and 2 provide validity and agreement of *declxfer*. The received message, d, containing a declaration vector, is unpacked and merged with the local active sources vector:

$$\begin{aligned}
&\mathit{get\_convictions}(d, actv_d) : \mathit{active\_vector\_type}[D] =\\
&\quad \mathit{merge\_active}(\mathit{extract\_decvec}(\mathit{unpack\_vec}(d)), actv_d)
\end{aligned}$$

It is easy to prove that the properties *symmetric_agreement?*, *declaration_agreement?*, and *benign_declaring?* are preserved by *get_convictions* with a fixed argument d.

Conclusion

We have provided a complete formal model of two of SPIDER's protocols: Interactive Consistency and Diagnosis. We have given formal proofs in PVS that under the Dynamic Fault Assumption and the Correct Active Sources Assumption, Interactive Consistency satisfies Validity and Agreement, and that Diagnosis preserves the Dynamic Fault Assumption and the Correct Active Sources Assumption and moreover is able to declare some faulty nodes.

The design effort and the parallel Formal Verification effort showed a remarkable cross-fertilization. The *symmetric_agreement?* property was first introduced as a fix to the formal verification. Later it turned out to be an essential premise. Considerations for readmission of transiently faulty nodes showed that Step 2 in the Interactive Consistency Protocol effectively prevented a recovering node from ever being readmitted. The ensuing redesign uncovered the potential to straighten the PVS code of the Interactive Consistency Protocol. That in turn lead to the presentation of Interactive Consistency as a combination of single source broadcast and consistent source exchange, the latter being reused for declaration exchange. The discovery that the Maximum Fault Assumption was unsuitable for readmission led to a complete redesign that culminated in the failure of agreement of Interactive Consistency and the recognized need for Step 5.

Probably the most important lesson that we can draw is the following: If our conjecture that all accusations against like nodes may be turned into declarations turns out true, then the Diagnosis Protocol can be simplified. The *accused* value for trust values against like nodes, and the exchange of accusations against like nodes can be scrapped. This saves $N \times N + M \times M$ register bits and half the bandwidth of the accusation exchange.

References

1. Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington Section.
2. Paul S. Miner, Mahyar Malekpour, and Wilfredo Torres-Pomales. Conceptual design of a Reliable Optical BUS (ROBUS). In *Digital Avionics Systems Conference DASC*, October 2002. To Appear.
3. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
4. John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September/October 1999.
5. John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001. Available at http://www.csl.sri.com/~rushby/abstracts/buscompare.
6. T. Basil Smith. Fault tolerant processor concepts and operations. In *Fault Tolerant Computing Symposium 14*, pages 158–163. IEEE Computer Society, 1984.
7. Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Reliable Distributed Systems Symposium*, pages 1–8, October 1988.
8. C. J. Walter, R. M. Kieckhafer, and A. M. Finn. MAFT: A multicomputer architecture for fault-tolerance in real-time control systems. In *IEEE Real-Time Systems Symposium*, December 1985.
9. Chris J. Walter, Patrick Lincoln, and Neeraj Suri. Formally verified on-line diagnosis. *IEEE Transactions on Software Engineering*, 23(11):684–721, November 1997.

Using HOL to Study *Sugar 2.0* Semantics

Michael J. C. Gordon

University of Cambridge Computer Laboratory
William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, U.K.
mjcg@cl.cam.ac.uk
http://www.cl.cam.ac.uk/~mjcg

Abstract. The Accellera standards-promoting organisation selected *Sugar 2.0*, IBM's formal specification language, as a standard that it says "will drive assertion-based verification". *Sugar 2.0* combines aspects of Interval Temporal Logic (ITL), Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) into a property language suitable for both formal verification and use with simulation test benches. As industrial strength languages go it is remarkably elegant, consisting of a small kernel conservatively extended by numerous definitions or 'syntactic sugar' (hence the name).
We are constructing a semantic embedding of *Sugar 2.0* in the version of higher order logic supported by the HOL system. To 'sanity check' the semantics we tried to prove some simple properties and as a result a few small bugs were discovered. We hope eventually to obtain a formal semantics that, with high confidence, matches the official 'golden' semantics issued by Accellera.
We are contemplating a variety of applications of the semantics, including building a semantics-directed Sugar model checker inside HOL. We also hope to investigate generating checkers by executing proof scripts that rewrite the semantics of particular constructs into an executable form. In the longer term we want to investigate the use of theorem proving to reason about models with infinite state spaces, which might involve developing extensions of *Sugar 2.0*.

1 Background on Accellera and Sugar

The Accellera organisation's website has their mission statement:

> *To improve designers' productivity, the electronic design industry needs a methodology based on both worldwide standards and open interfaces. Accellera was formed in 2000 through the unification of Open Verilog International and VHDL International to focus on identifying new standards, development of standards and formats, and to foster the adoption of new methodologies.*
> *Accellera's mission is to drive worldwide development and use of standards required by systems, semiconductor and design tools companies, which enhance a language-based design automation process. Its Board of Directors guides all the operations and activities of the organisation and is comprised of representatives from ASIC manufacturers, systems companies and design tool vendors.*

Faced with several syntactically and semantically incompatible formal property languages, Accellera initiated a process of selecting a standard property language to "drive assertion-based verification".
Four contributions were initially considered

- Motorola's CBV language;
- IBM's Sugar (the language of its RuleBase FV toolset);
- Intel's ForSpec;
- Verisity's *e* language (the language of the Specman Elite test-bench).

After a combination of discussion and voting, some details of which can be viewed on the web[1], attention was narrowed down to Sugar and CBV, and then in April 2002 a vote[2] selected IBM's submission, *Sugar 2.0*. *Sugar 2.0* is primarily an LTL-based language that is a successor to the CTL-based *Sugar 1* [1]. A key idea of both languages is the use of ITL-like [4] constructs called *Sugar Extended Regular Expressions*. *Sugar 2.0*

[1] http://www.eda.org/vfv/hm/
[2] http://www.eda.org/vfv/hm/0795.html

retains CTL constructs in its *Optional Branching Extension* (OBE), but this is de-emphasised in the defining document.

Besides moving from CTL to LTL, *Sugar 2.0* supports clocking and finite paths. Clocking allows one to specify on which clock edges signals are sampled. The finite path semantics allows properties to be interpreted on simulation runs, as in test-bench tools like Vera and Specman[3]

The addition of clocking and finite path semantics makes the *Sugar 2.0* semantics more than twice as complicated as the *Sugar 1* semantics. However, for a real 'industry standard' language *Sugar 2.0* is still remarkably simple, and it was routine to define the abstract syntax and semantics of the whole language in the logic of the HOL system [3].

In Section 2 we discuss the point of embedding Sugar in HOL. In Section 3, semantic embedding is reviewed and illustrated on simplified semantics of fragments of *Sugar 2.0*. In Section 4, the semantics of full *Sugar 2.0* is discussed, including finite paths and clocking. Due to space limitations, the complete semantics of *Sugar 2.0* is not given here, but can be found on the web.[4] In Section 5, progress so far in analysing the semantics using the HOL system is discussed. Finally, there is a short section of conclusions.

2 Why embed Sugar in HOL?

There are several justifications for the work described here. This project started in April 2002 and its goals are still being defined. Current motivations include the following.

2.1 Sanity checking and proving meta-theorems

By formalising the semantics and passing it through a parser and type-checker one achieves a first level of sanity checking of the definition. One also exposes possible ambiguities, fuzzy corner cases etc (e.g. see Section 4.2). The process is also very educational for the formaliser and a good learning exercise.

There are a number of meta-theorems one might expect to be true, and proving them with a theorem prover provides a further and deeper kind of sanity checking. In the case of *Sugar 2.0*, such meta-theorems include showing that expected simplifications to the semantics occur if there is no non-trivial clocking, that different semantics of clocking are equivalent and that if finite paths are ignored then the standard 'text-book semantics' results. Such meta-theorems are generally mathematically shallow, but full of tedious details i.e. ideal for automated theorem proving. See Section 5 for what we have proved so far.

2.2 Validating definitional extensions

A key feature of the Sugar approach – indeed the feature from which the name "Sugar" is derived – is to have a minimal kernel augmented with a large number of definitions – i.e. syntactic sugar – to enhance the usability (but not the expressive power) of the language.

The definitions can be validated by proving that they achieve the correct semantics. See the end of Section 5.3 for some examples.

2.3 Machine processable semantics

The current *Sugar 2.0* document is admirably clear, but it is informal mathematics presented as typeset text. Tool developers could benefit from a machine readable version. One might think of using some standard representation of mathematical content, like MathML[5], however there is currently not much mathematically sophisticated tool support for such XML-based representations. See the end of Section 5.4 for a bit more discussion.

Higher order logic is a widely used formalisation medium (versions of higher order logic are used by HOL, Isabelle/HOL, PVS, NuPrl and Coq) and the semantic embedding of model-checkable logics in HOL is standard [6,5]. Once one has a representation in higher order logic, then representations in other formats should be straightforward to derive.

[3] There is a 'Sugar2e' tool available from NoBug Consulting.

[4] http://www.cl.cam.ac.uk/~mjcg/Sugar/

[5] http://www.w3.org/Math/

2.4 Basis for research

We hope to develop semantically-based reasoning and checking infrastructure in HOL to support *Sugar 2.0*, and a prerequisite for this is to have a 'golden semantics' to which application-specific semantics can be proved equivalent.

We are interested in the development of property languages that support data operations and variables ranging over infinite data-types like numbers (e.g. including reals and complex numbers for DSP applications). Some sort of mixture of Hoare Logic and *Sugar 2.0* is being contemplated. Incrementally developing constructs by extending an existing semantics of *Sugar 2.0* is a way to ensure some backward compatibility with industry-standard language. Also, we might wish to prove sanity checking meta-theorem about our extended language, e.g. that it collapses to *Sugar 2.0* when there are no infinite types.

Sugar 2.0 is explicitly designed for use with simulation as well as formal verification. We are interested in using the HOL platform to experiment with combinations of execution, checking and theorem-proving. To this end we are thinking about implementing tools to transform properties stated in Sugar to checking automata. This is inspired by IBM's FoCs project[6], but uses compilation by theorem proving to ensure semantic equivalence between the executable checker and the source property.

2.5 Education

Both semantic embedding and property specification are taught as part of the Computer Science undergraduate course at Cambridge University, and being able to illustrate the ideas on a real example like *Sugar 2.0* is pedagogically valuable. Teaching an industrial property language nicely complements and motivates academic languages like ITL, LTL and CTL.

The semantic embedding of *Sugar 2.0* in the HOL system is an interesting case study. It illustrates some issues in making total functional definitions, and the formal challenges attempted so far provide insight into how to perform structural induction using the built-in tools. Thus *Sugar 2.0* has educational potential for training HOL users. In fact, the semantics described in this paper is an example distributed with HOL.[7]

3 Review of semantic embedding in higher order logic

Higher order logic is an extension of first-order predicate calculus that allows quantification over functions and relations. It is a natural notation for formalising informal set theoretic specifications (indeed, it is usually more natural than formal first-order set theories, like ZF). We hope that the HOL notation we use in what follows is sufficiently close to standard informal mathematics that it needs no systematic explanation.

We use Church's λ-notation for denoting functions: a 'lambda-term' like $\lambda x.\ t$, where x is a variable and t a term, denotes the function that maps a value v to the result of substituting v for the variable x in t (the infix notation $x \mapsto t$ is sometimes used instead of $\lambda x.\ t$). If P is a function that returns a truth-value (i.e. a predicate), then P can be thought of a set, and we write $x \in P$ to mean $P(x)$ is true. Note that $\lambda x.\ \cdots x \cdots$ corresponds to the set abstraction $\{x \mid \cdots x \cdots\}$. We write $\forall x \in P.\ Q(x)$, $\exists x \in P.\ Q(x)$ to mean $\forall x.\ P(x) \Rightarrow Q(x)$, $\exists x.\ P(x) \wedge Q(x)$, respectively.

To embed[8] a language in HOL one first defines constructors for all the syntactic constructs of the language. This is the 'abstract syntax' and provides a representation of parse trees as terms in the logic. The semantics is then specified by defining a semantic function that recursively maps each construct to a representation of its meaning.

For *Sugar 2.0*, a model $\mathtt{M}$ is a quintuple $(\mathtt{S}_\mathtt{M}, \mathtt{S0}_\mathtt{M}, \mathtt{R}_\mathtt{M}, \mathtt{P}_\mathtt{M}, \mathtt{L}_\mathtt{M})$, where $\mathtt{S}_\mathtt{M}$ is a set of states, $\mathtt{S0}_\mathtt{M}$ is the subset of initial states, $\mathtt{R}_\mathtt{M}$ is a transition relation (so $\mathtt{R}_\mathtt{M}(s, s')$ means s' is a possible successor state to s), $\mathtt{P}_\mathtt{M}$ is a set of atomic propositions, and $\mathtt{L}_\mathtt{M}$ is a valuation that maps a state to the set of atomic propositions that hold at the state (so $\mathtt{L}_\mathtt{M}\ s\ \mathtt{p}$ is true iff atomic proposition $\mathtt{p}$ is true in state s).

[6] http://www.haifa.il.ibm.com/projects/verification/focs/
[7] http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/hol/hol98/examples/Sugar2/
[8] We shall only be concerned with so called 'deep embeddings' here [2].

3.1 Boolean expressions in Sugar

The syntax of boolean expressions (ranged over by b, b_1, b_2 etc.) is built from atomic propositions (ranged over by p) using negation ($\neg$) and conjunction ($\wedge$):

$$\begin{array}{lll} b ::= & p & \text{(Atomic formula)} \\ & \mid \neg b & \text{(Negation)} \\ & \mid b_1 \wedge b_2 & \text{(Conjunction)} \end{array}$$

This is defined in HOL by a recursive type definition of a type that represents the syntax of boolean expressions. Other boolean expressions are added via definitions (e.g. see Section 5.3 for the definition of disjunction: $b_1 \vee b_2$).

Let `l` range over predicates on P_M, called "truth assignments" in the Sugar documentation. The semantics of boolean expressions is given by defining a semantic function `B_SEM` such that `B_SEM M l b` if true iff `b` is built from propositions in P_M and it is true with respect to the truth assignment `l`.

If we write (`M, l` $\models$ `b`) for `B_SEM M l b` then the semantics is given by

```
((M, l ⊨ p)       = p ∈ P_M ∧ p ∈ l)
∧
((M, l ⊨ T)       = T)
∧
((M, l ⊨ ¬b)      = ¬(M, l ⊨ b))
∧
((M, l ⊨ b1∧b2) = (M, l ⊨ b1) ∧ (M, l ⊨ b2))
```

Note that the symbol $\wedge$ is overloaded: the first occurrence in the equation above is part of the boolean expression syntax of Sugar, but the second occurrence is higher order logic.

Before looking at the full semantics of *Sugar 2.0*, we first consider a simplified semantics in which there is no clocking, and paths are always infinite. We consider separately the parts of *Sugar 2.0* corresponding to Interval Temporal Logic (ITL), Linear Temporal Logic (LTL) and Computation Tree Logic (CTL).

3.2 ITL: Sugar Extended Regular Expressions (SEREs)

Interval Temporal Logic (ITL) provides formulas that are true or false of intervals of states. Here we just consider finite intervals, though recent formulations of ITL[9] allow intervals to be infinite. For Sugar we only need to consider ITL formulas, as there are no constructs corresponding to ITL expressions (expressions map intervals to values). Providing more elaborate ITL constructs in Sugar strikes us as an interesting research topic.

The Sugar subset corresponding to ITL is called *Sugar Extended Regular Expressions* (SEREs). If r, r_1, r_2 etc. range over SEREs and p ranges over the set P_M of atomic propositions, then the syntax is given by:

$$\begin{array}{lll} r ::= & p & \text{(Atomic formula)} \\ & \mid \{r_1\} \mid \{r_2\} & \text{(Disjunction)} \\ & \mid r_1 \;;\; r_2 & \text{(Concatenation)} \\ & \mid r_1 : r_2 & \text{(Fusion: ITL's chop)} \\ & \mid \{r_1\} \;\&\&\; \{r_2\} & \text{(Length matching conjunction)} \\ & \mid \{r_1\} \;\&\; \{r_2\} & \text{(Flexible matching conjunction)} \\ & \mid r[*] & \text{(Repeat)} \end{array}$$

The semantics of SEREs is given by defining a semantic function `S_SEM` such that `S_SEM M w r` if true iff `w` is in the language of the extended regular expression `r`. We write (`M, w` $\models$ `r`) for `S_SEM M w r`.

If `wlist` is a list of lists then `Concat wlist` is the concatenation of the lists in `wlist` and if P is some predicate then `Every` P `wlist` means that P(`w`) holds for every `w` in `wlist`.

The semantics `S_SEM M w r` is defined in HOL by recursion on `r`.

[9] http://www.cms.dmu.ac.uk/~cau/itlhomepage/

```
((M, w ⊨ b) =
  ∃l. (w = [l]) ∧ (M, l ⊨ b))
∧
((M, w ⊨ r1;r2) =
  ∃w1 w2. (w = w1w2) ∧ (M. w1 ⊨ r1) ∧ (M, w2 ⊨ r2))
∧
((M, w ⊨ r1:r2) =
  ∃w1 w2 l. (w = w1 [l] w2) ∧
            (M, (w1 [l]) ⊨ r1) ∧ (M, ([l] w2) ⊨ r2))
∧
((M, w ⊨ {r1}|{r2}) =
  (M, w ⊨ r1) ∨ (M, w ⊨ r2))
∧
((M, w ⊨ {r1}&&{r2}) =
  (M, w ⊨ r1) ∧ (M, w ⊨ r2))
∧
((M, w ⊨ {r1}&{r2}) =
  ∃w1 w2. (w = w1w2) ∧
          (((M, w ⊨ r1) ∧ (M, w1 ⊨ r2))
           ∨
           ((M, w ⊨ r2) ∧ (M, w1 ⊨ r1))))
∧
((M, w ⊨ r[*]) =
  ∃wlist. (w = Concat wlist) ∧ Every (λw. (M. w ⊨ r)) wlist)
```

This definition is manifestly primitive-recursive, and so is automatically proved total by HOL [7]. The intuitive semantics of SERE's is explained in the *Sugar 2.0* documentation [8].

3.3 LTL: Sugar Foundation Language (FL)

Sugar 2.0 has a kernel combining standard LTL notation with a less standard `abort` operation and some constructs using SEREs. The suffix "!" found on some constructs indicates that these are 'strong' (i.e. liveness-enforcing) operators. The distinction between strong and weak operators is discussed and motivated in the *Sugar 2.0* literature (e.g. [9, Section 4.11]).

f ::= p	(Atomic formula)
| ¬f	(Negation)
| $f_1 \wedge f_2$	(Conjunction)
| X!f	(Successor)
| [f_1 U f_2]	(Until)
| {r}(f)	(Suffix implication)
| {r_1} |-> {r_2}!	(Strong suffix implication)
| {r_1} |-> {r_2}	(Weak suffix implication)
| f abort b	(Abort)

Numerous additional notations are introduced as syntactic sugar. These are easily formalised as definitions in HOL. Some examples are given in Section 5.3.

Being LTL, the semantics of FL formulas is defined with respect to a path π, which (in the simplified semantics here) is a function from the natural numbers to states.

We define a semantic function `F_SEM` such that `F_SEM M` π `f` means FL formula `f` is true of path π. We write (M, π ⊨ r) for `F_SEM M` π `f`.

Note that in the semantics below it is not assumed that paths π are necessarily computations of `M` (i.e. satisfy `Path M` π, as defined in Section 3.4). This is important for the `abort` construct (where the $\exists\pi'$ quantifies over all paths).

The notation π_i denotes the i-th state in the path (i.e. $\pi(i)$); π^i denotes the 'i-th tail' of π – the path obtained by chopping i elements off the front of π (i.e. $\pi^i = \lambda n.\ \pi(n+i)$); $\pi^{(i,j)}$ denotes the finite sequence of states from i to j in π, i.e. $\pi_i\pi_{i+1}\cdots\pi_j$. The juxtaposition $\pi^{(i,j)}\pi'$ denotes the path obtained by concatenating the finite sequence $\pi^{(i,j)}$ on to the front of the path π'.

The function $\hat{L}_M$ denotes the point-wise extension of L_M to finite sequences of states (i.e. `MAP` L_M in HOL and functional programming notation).

The definition of `F_SEM M` π `f` is by recursion on `f`.

```
((M, π ⊨ b) = (M, L_M(π_0) ⊨ b))
∧
((M, π ⊨ ¬f) = ¬(M, π ⊨ f))
∧
((M, π ⊨ f1 ∧ f2) = (M, π ⊨ f1) ∧ (M, π ⊨ f2))
∧
((M, π ⊨ X! f) = (M, π^1 ⊨ f))
∧
((M, π ⊨ [f1 U f2]) =
  ∃k. (M, π^k ⊨ f2) ∧ ∀j. j < k ⇒ (M, π^j ⊨ f1))
∧
((M, π ⊨ {r}(f)) =
  ∀j. (M, (L̂_M (π^(0,j))) ⊨ r) ⇒ (M, π^j ⊨ f))
∧
((M, π ⊨ {r1}|->{r2}!) =
  ∀j. (M, (L̂_M (π^(0,j))) ⊨ r1)
       ⇒ ∃k. j ≤ k ∧ (M, (L̂_M (π^(j,k))) ⊨ r2))
∧
((M, π ⊨ {r1}|->{r2}) =
  ∀j. (M, (L̂_M (π^(0,j))) ⊨ r1)
       ⇒ (∃k. j ≤ k ∧ (M, (L̂_M (π^(j,k))) ⊨ r2))
          ∨
          ∀k. j ≤ k ⇒ ∃w. (M, (L̂_M (π^(j,k)))w ⊨ r2))
∧
((M, π ⊨ f abort b) =
  ((M, π ⊨ f )
   ∨
   ∃j π'. (M, π^j ⊨ b) ∧ (M, π^(0,j-1)π' ⊨ f)))
```

In this semantics, paths π are infinite, as in the classical semantics of LTL for model checking. A version that also handles finite paths, suitable for evaluation on simulation runs, is given in Section 4.2.

3.4 CTL: Sugar Optional Branching Extension (OBE)

The syntax of the *Sugar 2.0* OBE is completely standard. The syntax of the OBE formulas is:

$\mathtt{f} ::= \mathtt{p}$	(Atom)
$\mid \neg\mathtt{f}$	(Negation)
$\mid \mathtt{f}_1 \wedge \mathtt{f}_2$	(Conjunction)
$\mid$ **EX**f	(Some successors)
$\mid$ **E**[f$_1$ **U** f$_2$]	(Until – along some path)
$\mid$ **EG**f	(Always on some path)

For the semantics, define `Path M` π to be true iff π is a computation of `M`:

```
Path M $\pi$ = $\forall$n. $R_M(\pi_n, \pi_{n+1})$
```

The semantic function O_SEM is defined so that O_SEM M s f is true iff f is true of M at state s. Write (M, s $\models$ f) for O_SEM M s f, which is defined by recursion on f by:

```
((M, s ⊨ b) = (M, L_M(s) ⊨ b))
∧
((M, s ⊨ ¬f) = ¬(M, s ⊨ f))
∧
((M, s ⊨ f1∧f2) = (M, s ⊨ f1) ∧ (M, s ⊨ f2))
∧
((M, s ⊨ EX f) =
  ∃π. Path M π ∧ (π_0 = s) ∧ (M, π_1 ⊨ f))
∧
((M, s ⊨ [f1 U f2]) =
  ∃π. Path M π ∧ (π_0 = s) ∧
      (M, π_k ⊨ f2) ∧ ∀j. j < k ⇒ (M, π_j ⊨ f1))
∧
((M, s ⊨ EG f) =
  ∃π. Path M π ∧ (π_0 = s) ∧ ∀j. (M, π_j ⊨ f))
```

4 Full *Sugar 2.0* semantics in higher order logic

The full *Sugar 2.0* language extends the constructs described above with the addition of clocking and support for finite paths.

The clocking constructs allow (possibly multiple) clocks to be declared, see Section 4.1. Clocks define when signals are sampled, so the next value of a signal s with respect to a clock c is the value of s at the next rising edge of !c.

Simulators compute finite executions of a model, so to support checking whether a property holds over such a simulation run, *Sugar 2.0* defines the meaning of each construct on both finite and infinite paths.

Adding clocks and finite paths greatly complicates the language, though it is still surprisingly elegant.

We have formalised the full semantics of *Sugar 2.0* via a deep embedding in higher order logic. Corresponding to Appendix A.1 of the *Sugar 2.0* specification submitted to Accellera [9] we have defined types bexp, sere, fl and obe in the HOL logic to represent the syntax of Boolean Expressions, Sugar Extended Regular Expressions (SEREs), formulas of the Sugar Foundation Language (FL) and formulas of the Optional Branching Extension (OBE), respectively.

Corresponding to Appendix A.2 of the Sugar documentation we have defined semantic functions B_SEM, S_SEM, F_SEM and O_SEM that interpret boolean expressions, SEREs, FL formulas and OBE formulas, respectively.

Due to space constraints we do not give the semantics here, but full details are available on the web at:

http://www.cl.cam.ac.uk/~mjcg/Sugar

The semantics is evolving and we hope to keep the HOL version up to date with respect to the official version.
In the next two sub-sections we discuss clocking and finite paths.

4.1 Clocking

If b is a boolean expression, then the SERE b@c recognises a sequence of states in which b is true on the next rising edge of c. Thus b@c behaves like {¬c[*] ; c ∧ b}.

More generally, if r is a SERE and c a variable then r@c is a SERE in which all variables inside r are clocked with respect to the rising edges of c.

The semantics of clocked SEREs can be given in two ways:

1. by making a clocking context part of the semantic function, i.e. defining $(\texttt{M, w} \overset{\texttt{c}}{\models} \texttt{r})$ instead of the unclocked $(\texttt{M, w} \models \texttt{r})$;
2. by translating clocked SEREs into unclocked SEREs using rewriting rules.

With the first approach (1), which is taken as the definition in the Accellera report, one defines

$$
\begin{array}{ll}
(\texttt{M, w} \overset{\texttt{c}}{\models} \texttt{b}) = & \\
\quad \exists \texttt{n}.\ \texttt{n} \geq 1 & \wedge \\
\qquad (\text{length } \texttt{w} = \texttt{n}) & \wedge \\
\qquad (\forall \texttt{i}.\ 1 \leq \texttt{i} \wedge \texttt{i} < \texttt{n} \Rightarrow (\texttt{M}, \texttt{w}_{\texttt{i}-1} \models \neg\texttt{c}) & \wedge \\
\qquad (\texttt{M}, \texttt{w}_{\texttt{n}-1} \models \texttt{c} \wedge \texttt{b}) &
\end{array}
$$

$$(\texttt{M, w} \overset{\texttt{c}}{\models} \texttt{r@c1}) = (\texttt{M, w} \overset{\texttt{c1}}{\models} \texttt{r})$$

together with equations like those in Section 3.2, but with $\overset{\texttt{c}}{\models}$ replacing $\models$. Notice that an inner clock overrides an outer clock (i.e. `c1` is used to clock variables inside `r` in `r@c1`: the clock context `c` is overridden by `c1` inside `r`).

The second approach (2) is to translate clocked SEREs to unclocked SEREs using rewrites

`b@c`	$\longrightarrow$	`{¬c[*]; c∧b}`
`{r1;r2}@c`	$\longrightarrow$	`{r1@c};{r2@c}`
`{r1:r2}@c`	$\longrightarrow$	`{r1@c}:{r2@c}`
`{{r1}|{r2}}@c`	$\longrightarrow$	`{r1@c}|{r2@c}`
`{{r1}&&{r2}}@c`	$\longrightarrow$	`{r1@c}&&{r2@c}`
`{{r1}&{r2}}@c`	$\longrightarrow$	`{r1@c}&{r2@c}`
`r[*]@c`	$\longrightarrow$	`{r@c}[*]`
`r@c1@c`	$\longrightarrow$	`r1@c1`

these rewrites cannot be taken as equational definitions, but need to be applied from the outside in: e.g. one must rewrite `b@c1@c` to `b@c1` (eliminating `c`) rather than rewriting the sub-term `b@c1` first, resulting in `{¬c1[*]; c1∧b}@c`. We have proved the two semantics for clocking SEREs are equivalent, see Section 5.3. One can also clock formulas, `f@c`, and there may be several clocks. Consider:[10]

```
G(req_in -> X!(req_out@cb))@ca
```

this means that the entire formula is clocked on clock `ca`, except that signal `req_out` is clocked on `cb`. Clocks do not 'accumulate', so the signal `req_out` is only clocked by `cb`, not by both clocks. Thus `cb` 'protects' `req_out` from the main clock, `ca`, i.e.:

```
req_out@cb@ca = req_out@cb
```

As with the clocking of SEREs, this meaning of clocking prevents us simply defining:

```
req_out@cb = [¬cb U (cb ∧ req_out)]
```

since if this were the definition of `req_out@cb` then we would be forced to have:

```
req_out@cb@ca = [¬cb U (cb ∧ req_out)]@ca
```

when we actually want

```
req_out@cb@ca = req_out@cb
```

Thus, as with SEREs, we cannot just rewrite away clocking constructs using equational reasoning, but if one starts at the outside and works inwards, then one can systematically compile away clocking. The rules for doing this are given in the *Sugar 2.0* Accellera documentation as part of the implementation of formal verification [9, Appendix B.1]. We are currently in the process of trying to validate the clocking rewrites, see Section 5.3.

[10] The discussion of clocking here is based on email communication with Cindy Eisner.

The official semantics uses the approach like (1) above of having the currently active clock as an argument to the semantic function for formulas. In fact two semantics are given: one for 'weak' clocking and one for 'strong' clocking. The weak clocking is specified in HOL by defining

$(\mathtt{M},\ \pi \overset{\mathtt{c}}{\models} \mathtt{f})$

and the strong clocking by defining

$(\mathtt{M},\ \pi \overset{\mathtt{c!}}{\models} \mathtt{f})$

We shall not give the complete semantics here (they are available on the web), but just show the semantics of boolean expressions b:

$$\begin{array}{l}((\mathtt{M},\ \pi \overset{\mathtt{c}}{\models} \mathtt{b}) = \\ \quad \forall \mathtt{i} \in \mathtt{pl}\,\pi.\ (\mathtt{M},\ (\hat{\mathtt{L}}_{\mathtt{M}}\ (\pi^{(0,\mathtt{i})})) \overset{\mathtt{T}}{\models} \neg\mathtt{c[*];c}) \Rightarrow (\mathtt{M},\ \mathtt{L}_{\mathtt{M}}(\pi_{\mathtt{i}}) \models \mathtt{b}))\end{array}$$

This says that *if* there is a first rising edge of c at time i, then b is true at i.

$$\begin{array}{l}((\mathtt{M},\ \pi \overset{\mathtt{c!}}{\models} \mathtt{b}) = \\ \quad \exists \mathtt{i} \in \mathtt{pl}\,\pi.\ (\mathtt{M},\ (\hat{\mathtt{L}}_{\mathtt{M}}\ (\pi^{(0,\mathtt{i})})) \overset{\mathtt{T}}{\models} \neg\mathtt{c[*];c}) \wedge (\mathtt{M},\ \mathtt{L}_{\mathtt{M}}(\pi_{\mathtt{i}}) \models \mathtt{b}))\end{array}$$

This says that *there is* a first rising edge, and if it occurs at time i, then b is true at i. Thus the strongly clocked semantics assumes the clock is 'live', but the weakly clocked semantics doesn't (compare the concepts of total and partial correctness).

4.2 Finite paths

Sugar 2.0 gives a semantics to formulas for both finite and infinite paths. To represent this, we model a path as being either a non-empty[11] finite list of states or a function from natural numbers to states and define a predicate **finite** to test if a path is a finite list. The function **length** gives the length of a finite path (it is not defined on paths for which **finite** is not true).

We interpret the official semantics locution

"for every $j <$ **length**(π): $\cdots j \cdots$"

as meaning

"for every j: (**finite** π implies $j <$ **length** π) implies $\cdots j \cdots$"

and we interpret the official semantics locution

"there exists $j <$ **length**(π) s.t. $\cdots j \cdots$"

as meaning

"there exists j s.t. (**finite** π implies $j <$ **length** π) and $\cdots j \cdots$"

Define **pl** π n to mean that if π is finite then n is less than the length of π, i.e. the predicate **pl** is defined by

$\mathtt{pl}\ \pi\ n\ =\ \mathtt{finite}\ \pi\ \Rightarrow\ n < \mathtt{length}\ \pi$

We can then write "$\forall i \in \mathtt{pl}\ \pi.\ \cdots i \cdots$" and "$\exists i \in \mathtt{pl}\ \pi.\ \cdots i \cdots$" for the locutions above. The name "pl" is short for "path length"

Here is a version of the unclocked FL semantics that allows paths to be finite.

$$\begin{array}{l}((\mathtt{M},\ \pi \models \mathtt{b}) = (\mathtt{M},\ \mathtt{L}_{\mathtt{M}}(\pi_0) \models \mathtt{b})) \\ \wedge \\ ((\mathtt{M},\ \pi \models \neg\mathtt{f}) = \neg(\mathtt{M},\ \pi \models \mathtt{f})) \\ \wedge \\ ((\mathtt{M},\ \pi \models \mathtt{f1} \wedge \mathtt{f2}) = (\mathtt{M},\ \pi \models \mathtt{f1}) \wedge (\mathtt{M},\ \pi \models \mathtt{f2})) \\ \wedge \\ ((\mathtt{M},\ \pi \models \mathtt{X!\,f}) = \mathtt{pl}\ \pi\ 1 \wedge (\mathtt{M},\ \pi^1 \models \mathtt{f}))\end{array}$$

[11] The need for finite paths to be non-empty arose when trying to prove some properties. This requirement does not seem to be explicit in the Accellera specification.

```
  $\wedge$
  ((M, $\pi$ $\models$ [f1 U f2]) =
    $\exists$k $\in$ pl $\pi$.
     (M, $\pi^{k}$ $\models$ f2) $\wedge$ $\forall$j $\in$ pl $\pi$. j < k $\Rightarrow$ (M, $\pi^{j}$ $\models$ f1))
  $\wedge$
  ((M, $\pi$ $\models$ {r}(f)) =
    $\forall$j $\in$ pl $\pi$. (M, ($\hat{L}_M$ ($\pi^{(0,j)}$)) $\models$ r) $\Rightarrow$ (M, $\pi^{j}$ $\models$ f))
  $\wedge$
  ((M, $\pi$ $\models$ {r1}|->{r2}!) =
    $\forall$j $\in$ pl $\pi$. (M, ($\hat{L}_M$ ($\pi^{(0,j)}$)) $\models$ r1)
              $\Rightarrow$ $\exists$k $\in$ pl $\pi$. j $\leq$ k $\wedge$ (M, ($\hat{L}_M$ ($\pi^{(j,k)}$)) $\models$ r2))
  $\wedge$
  ((M, $\pi$ $\models$ {r1}|->{r2}) =
    $\forall$j $\in$ pl $\pi$. (M, ($\hat{L}_M$ ($\pi^{(0,j)}$)) $\models$ r1)
              $\Rightarrow$ ($\exists$k $\in$ pl $\pi$. j $\leq$ k $\wedge$ (M, ($\hat{L}_M$ ($\pi^{(j,k)}$)) $\models$ r2))
                 $\vee$
                 $\forall$k $\in$ pl $\pi$. j $\leq$ k $\Rightarrow$ $\exists$w. (M, ($\hat{L}_M$ ($\pi^{(j,k)}$))w $\models$ r2))
  $\wedge$
  ((M, $\pi$ $\models$ f abort b) =
    ((M, $\pi$ $\models$ f )
     $\vee$
     $\exists$j $\in$ pl $\pi$.
      0 < j $\wedge$ $\exists\pi$'. (M, $\pi^{j}$ $\models$ b) $\wedge$ (M, $\pi^{(0,j-1)}\pi'$ $\models$ f)))
```

This semantics has evolved from an existing unpublished semantics[12] of unclocked FL formulas.

5 Progress on analysing the semantics

We have established a number of properties of the semantics using the HOL system. Some of these went through first time without any problems, but others revealed bugs both in the *Sugar 2.0* semantics and original HOL representation of the semantics.

5.1 Characterising adjacent rising edges

Define:

```
FirstRise M $\pi$ c i     = (M, ($\hat{L}_M$ ($\pi^{(0,i)}$)) $\overset{T}{\models}$ $\neg$c[*];c)
NextRise M $\pi$ c (i,j) = (M, ($\hat{L}_M$ ($\pi^{(i,j)}$)) $\overset{T}{\models}$ $\neg$c[*];c)
```

The right hand sides of these definition occur in the *Sugar 2.0* semantics. We have proved that the definitions of `FirstRise` and `NextRise` give them the correct meaning, namely `FirstRise M` π `c i` is true iff `i` is the time of the first rising edge of `c`, and `NextRise M` π `c (i,j)` is true iff `j` is the time of the first rising edge of `c` after `i`.

```
$\vdash$ FirstRise M $\pi$ c i =
   ($\forall$j. j < i $\Rightarrow$ $\neg$(M, $L_M(\pi_j)$ $\models$ c)) $\wedge$ (M, $L_M(\pi_i)$ $\models$ c)

$\vdash$ i $\leq$ j
   $\Rightarrow$
  (NextRise M $\pi$ c (i,j) =
    ($\forall$k. i $\leq$ k $\wedge$ k < j $\Rightarrow$ $\neg$(M, $L_M(\pi_k)$ $\models$ c)) $\wedge$ (M, $L_M(\pi_j)$ $\models$ c))
```

[12] Personal communication from Cindy Eisner.

The proof of these were essentially routine, though quite a bit more tricky than expected. Immediate corollaries are

$\vdash$ `FirstRise` M π T i = (i = 0)
$\vdash$ i $\leq$ j $\Rightarrow$ (`NextRise` M π T (i,j) = (i = j))

5.2 Relating the clocked and unclocked semantics

If we define `ClockFree r` to mean that `r` contains no clocking constructs (a simple recursion over the syntax of SEREs), then clocking with T is equivalent to the unclocked SERE semantics.

$\vdash \forall$r. `ClockFree r` $\Rightarrow$ ((M, w $\overset{T}{\models}$ r) = (M, w $\models$ r))

The proof of this is an easy structural induction, and shows that when the clock is T, the clocked semantics of SEREs collapses to the semantics in Section 3.2.
We tried to prove a similar result for FL formulas, but at first this turned out to be impossible. The reason was that the proof required first showing

$\forall$f π. (M, π $\overset{T}{\models}$ f) = (M, π $\overset{T!}{\models}$ f))

However, the original semantics had the following:

(M, π $\overset{c!}{\models}$ b) = $\exists$i. `FirstRise` M π c i $\wedge$ (M, $L_M(\pi_i)$ $\models$ b)

(M, π $\overset{c}{\models}$ b) = $\exists$i. `FirstRise` M π c i $\Rightarrow$ (M, $L_M(\pi_i)$ $\models$ b)

Instantiating c to T and using the corollary about `FirstRise` yields

(M, π $\overset{T!}{\models}$ b) = $\exists$i. (i=0) $\wedge$ (M, $L_M(\pi_i)$ $\models$ b)

(M, π $\overset{T}{\models}$ b) = $\exists$i. (i=0) $\Rightarrow$ (M. $L_M(\pi_i)$ $\models$ b)

With this, clearly (M, π $\overset{T}{\models}$ b) is not equal to (M, π $\overset{T!}{\models}$ b). The solution, suggested by Cindy Eisner, is to replace the weak semantics by

(M, π $\overset{c}{\models}$ b) = $\forall$i. `FirstRise` M π c i $\Rightarrow$ (M, $L_M(\pi_i)$ $\models$ b)

so that we get

(M, π $\overset{T!}{\models}$ b) = $\exists$i. (i=0) $\wedge$ (M, $L_M(\pi_i)$ $\models$ b)

(M, π $\overset{T}{\models}$ b) = $\forall$i. (i=0) $\Rightarrow$ (M. $L_M(\pi_i)$ $\models$ b)

which makes (M, π $\overset{T}{\models}$ b) equal to (M, π $\overset{T!}{\models}$ b). The same change of $\exists$ to $\forall$ is also needed for the semantics of weak clocking for f1 $\wedge$ f2, X! f, {r}(f), {r1}|->{r2} and `f abort b`. With these changes, we used structural induction to prove:[13]

$\vdash \forall$f π. (M, π $\overset{T}{\models}$ f) = (M, π $\overset{T!}{\models}$ f)

However, we were still unable to prove

$\vdash \forall$f. `ClockFree f` $\Rightarrow$ ((M, π $\overset{T}{\models}$ f) = (M, π $\models$ f))

where here `ClockFree f` means that `f` contains no clocked FL formulas or SEREs. The proof attempt failed because the unclocked semantics for `[f1 U f2]` had a path length check, but the strongly clocked semantics didn't. After restricting the quantification of a variable in the strongly clocked semantics to values satisfying `pl` π, the proof went through.

[13] See Section 5.4 for further developments!

5.3 Validating the clock implementation rewriting rules

As discussed in Section 4.1, the semantics of clocked SEREs and formulas can be given in two ways:

1. by defining $\overset{c}{\models}$ and, for formulas, $\overset{c!}{\models}$;
2. by translating away clocking constructs `r@c`, `f@c` and `f@c!` using rewrites, then using the unclocked semantics $\models$.

The representation in HOL of the direct semantics (1) has already been discussed.
The definition of the translation (2) in HOL is straightforward: one just defines recursive functions `SClockImp`, that takes a clock and a SERE and returns a SERE, and `FClockImp` that takes a clock context and a formula and returns a formula. Thus roughly[14]

```
SClockImp : clock → sere → sere
FClockImp : clock →  fl  →  fl
```

We can then attempt to prove that

$\vdash$ ∀`r w c. (M, w` $\overset{c}{\models}$ `r) = (M, w` $\models$ `SClockComp c r)`

which turns out to be a routine proof by structural induction on `r`. However, the results for formulas

$\vdash$ ∀`f` π `c. (M,` π $\overset{c}{\models}$ `f) = (M,` π $\models$ `FClockComp c f)`

$\vdash$ ∀`f` π `c. (M,` π $\overset{c!}{\models}$ `f) = (M,` π $\models$ `FClockComp c! f)`

are harder, and we have not yet finished proving these (as of 5 July 2002). To see the complexity involved consider the rewrite for weakly clocked conjunctions [9, page 67]:

```
(f1 ∧ f2)@c    ⟶   [¬c W (c ∧ (f1@c ∧ f2@c))]
```

where `W` is the 'weak until' operator which is part of the definitional extension (i.e. syntactic sugar) defined as part of *Sugar 2.0*, namely:

```
[f1 W f2]  =  [f1 U f2] ∨ G f1
```

where `U` is a primitive (part of the kernel) but ∨ and `G` are defined by:

```
f1 ∨ f2  =  ¬(¬f1 ∧ ¬f2)
G f      = ¬F(¬f)
```

and `F` is defined by

```
F f  =  [T U f]
```

Let us define

`FClockCorrect M f` = (∀π `c. (M,` π $\overset{c}{\models}$ `f) = (M,` π $\models$ `FClockComp c f))`
∧
(∀π `c. (M,` π $\overset{c!}{\models}$ `f) = (M,` π $\models$ `FClockComp c! f))`

It is relatively straightforward to prove the cases for boolean formulas `b` and negations `¬f`, namely:

$\vdash$ ∀`M. FClockCorrect M b`

$\vdash$ ∀`M f. FClockCorrect M f` ⇒ `FClockCorrect M (¬f)`

For formula conjunction we want to prove:

∀`M f1 f2. FClockCorrect M f1 ∧ FClockCorrect M f2` ⇒ `FClockCorrect M (f1 ∧ f2)`

where the first ∧ is in higher order logic and the one in `f1 ∧ f2` is part of the Sugar formula syntax.

[14] We are glossing over details here, like what the type `clock` exactly is.

We got bogged down in details when we tried to prove this directly, so we first established some lemmas about $\vee$ and the unclocked semantics of the defined operators W, G and F.

$$\vdash \texttt{(M, }\pi \models \texttt{f1} \vee \texttt{f2)} = \texttt{(M, }\pi \models \texttt{f1)} \vee \texttt{(M, }\pi \models \texttt{f2)}$$

$$\vdash \texttt{(M, }\pi \models \texttt{F f)} = \exists \texttt{i} \in \texttt{pl}\,\pi.\ \texttt{(M, }\pi^{\texttt{i}} \models \texttt{f)}$$

$$\vdash \texttt{(M, }\pi \models \texttt{G f)} = \forall \texttt{i} \in \texttt{pl}\,\pi.\ \texttt{(M, }\pi^{\texttt{i}} \models \texttt{f)}$$

$$\vdash \neg\texttt{(M, }\pi \models \texttt{G f)} = \exists \texttt{i} \in \texttt{pl}\,\pi.\ \texttt{(M, }\pi^{\texttt{i}} \models \neg\texttt{f)}$$

$$\vdash \neg\texttt{(M, }\pi \models \texttt{G f)} = \exists \texttt{i} \in \texttt{pl}\,\pi.\ \texttt{(M, }\pi^{\texttt{i}} \models \neg\texttt{f)} \wedge \forall \texttt{j} \in \texttt{pl}\,\pi.\ \texttt{j} < \texttt{i} \Rightarrow \texttt{(M, }\pi^{\texttt{j}} \models \texttt{f)}$$

$$\vdash \texttt{(M, }\pi \models \texttt{[f1 W f2])} = \texttt{(M, }\pi \models \texttt{[f1 U f2])} \vee \texttt{(M, }\pi \models \texttt{G f1)}$$

Using these lemmas it is not too hard to prove the desired result about conjunctions. Besides helping with the proof of this, the lemmas also provide some sanity checking of the definitions.

5.4 Restricting quantifiers

The original semantics specifies that some of the quantifications over integer variables be restricted to range over values the are smaller than the length of the current path π (we represent this using $\texttt{pl}\,\pi$). Our initial attempts to relate the clocked and unclocked semantics needed additional quantifier restrictions to be added, as discussed at the end of Section 5.2 above. However, during email discussions with the *Sugar 2.0* designers it became clear that in fact all quantifications should be restricted, for otherwise the semantics would rely on the HOL logic's default interpretations of terms like π^j when π is finite and $j \geq \texttt{length}\ \pi$.[15] With HOL's default interpretation of 'meaningless' terms, it is unclear whether the semantics accurately reflects the designers intentions.

Thus the semantics was modified so that all quantifications are suitably restricted. In addition, and in the same spirit, we added the requirement that all terms $\pi^{(i,j)}$ occurred in a context where $i \leq j$, so that the arbitrary value of $\pi^{(i,j)}$ when $i > j$ was never invoked. Unfortunately these changes broke the proof of:

$$\vdash \forall \texttt{f}\ \pi.\ \texttt{(M, }\pi \overset{\texttt{T}}{\models} \texttt{f)} = \texttt{(M, }\pi \overset{\texttt{T!}}{\models} \texttt{f)}$$

and hence the proof relating the clocked and unclocked semantics. However, it turned out that there was a bug in the semantics: "$\texttt{l} > \texttt{k}$" occurred in a couple of places where there should have been "$\texttt{l} \geq \texttt{k}$", and when this change was made the proof of the above property, and the equivalence between the unclocked and true-clocked semantics, went through.

However, just as we thought everything was sorted out, the *Sugar 2.0* designers announced they had discovered a bug and pointed out that without their fix we should not have been able to prove what we had. This bug had arisen in the semantics of X! formulas when the $\exists$-to-$\forall$ change to the weakly clocked semantics (which we discussed in Section 5.2) was made.

Careful manual analysis showed that an error in the HOL semantics had been introduced when the $\exists$-to-$\forall$ change was made, and this error masked the bug that should have appeared when we tried to do the proof. Thus a bug in the HOL semantics allowed a proof to succeed when it shouldn't have! After removing the transcription error from the HOL semantics the proofs failed, as they should, and after the correct fix, supplied by the Sugar designers, was made to the semantics the proofs went through.

This experience with a transcription error masking a bug has sensitised us to the dangers of manually translating the typeset semantics into HOL. We had carefully and systematically manually checked that the HOL was a correct more than once, but nevertheless the error escaped detection. As a result, we are experimenting with ways of structuring LaTeX source to represent the 'deep structure' of the semantics rather than its 'surface form'. The idea is to define LaTeXcommands (macros) that are semantically meaningful and can be parsed directly into logic with a simple script. The LaTeX definitions of the commands will then

[15] The logical treatment of 'undefined' terms like 1/0 or hd[] has been much discussed. HOL uses a simple and consistent approach based on Hilbert's ε-operator. Other approaches include 'free logics' (i.e. logics with non-denoting terms) and three-valued logics in which formulas can evaluate to *true*, *false* and *undefined*.

generate the publication form of the semantics. By giving the commands extra parameters that can be used to hold strings for generating English, but ignored when translating to HOL, it appears possible to use LaTeX to represent the semantics. However, the resulting document source is rather complex and may be hard to maintain. The long term 'industry standard' solution to this problem is to use XML (e.g. MathML), but current infrastructure for MathML is either not quite ready (e.g. Publicon[16]) or not quite polished enough for everyday use (e.g. IBM texexplorer[17], Mozilla[18] and TtM[19])

6 Conclusions

It was quite straightforward to use the informal semantics in the *Sugar 2.0* documentation to create a deep embedding of the whole *Sugar 2.0* kernel. Attempting to prove some simple 'sanity checking' lemmas with a proof assistant quickly revealed bugs in the translated semantics (and possibly in the original). Further probing revealed more bugs.

It is hoped that the semantics in HOL that we now have is correct, but until further properties are proved we cannot be sure, and the experience so far suggests caution!

7 Acknowledgements

The *Sugar 2.0* team of Cindy Eisner and Dana Fisman patiently answered numerous email questions in great detail. They also supplied valuable comments and corrections to an earlier version of this paper, and suggested ways of modifying the HOL semantics to get the proofs described in Section 5 to go through.

References

1. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th International Conference on Computer Aided Verification (CAV)*, LNCS 2102. Springer-Verlag, 2001.
2. R. Boulton, A. Gordon, M. 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, Nijmegen, June 1992*, IFIP Transactions A-10, pages 129–156. North-Holland, 1992.
3. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.
4. J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In J. Diaz, editor, *Proceedings of the 10-th International Colloquium on Automata, Languages and Programming*, volume 154 of *LNCS*, pages 278 291. Springer Verlag, 1983.
5. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
6. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.
7. K. Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, Turku, Finland, August 1996: Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–397. Springer-Verlag, 1996.
8. `www.haifa.il.ibm.com/projects/verification/sugar/literature.html`.
9. `www.haifa.il.ibm.com/projects/verification/sugar/Sugar_2.0_Accellera.ps`.

[16] http://www.wolfram.com/products/publicon/
[17] http://www-3.ibm.com/software/network/techexplorer/
[18] http://www.mozilla.org/projects/mathml/
[19] http://hutchinson.belmont.ma.us/tth/mml/

Extending DOVE with Product Automata*

Elsa L. Gunter[1] and Yi Meng[1]

New Jersey Institute of Technology
elsa@cis.njit.edu
ym5@njit.edu
http://www.cis.njit.edu/~elsa

Abstract. With the continuing growth of computer systems including safety critical computer control systems, the need for reliable tools to help construct, analyze, and verify such systems also continues to grow. An example of such a tool is DOVE [1, 2]. One of the advantages of DOVE is that it combines the ease of use provide by a graphical user interface for describing specifications in the form of finite state machines with the rigor of proving linear temporal logic properties in a robust theorem prover, Isabelle [5]. In the work described in this paper we increase the utility of DOVE by extending it with the capability to build systems by specifying components.

1 Introduction

The need for effective assurance in the design of critical systems continues to grow as our dependence on such systems continues to spread and grow. In many cases, death or injury can be easily caused by any faults in these critical systems, such as safety critical systems. To try to address this need, there is a growing variety of formal methods tools. These tools are based on a variety of differing underlying methods. To be used outside the research community, such a tool must be fairly easy to use, and usually would require a graphical user interface. The aim of the Design Oriented Verification and Evaluation (DOVE) [1, 2], which was designed by the Defense Science and Technology Organization (DSTO) in Australia, is to provide a powerful tool to meet this challenge.

DOVE comprises three main components: the graphical editor for drawing finite state machine as specifications of systems, the animator for exploring various execution paths, and a prover, built on Isabelle [5], for verifying temporal logic properties of state machine. DOVE combines the ease of use afforded by a graphical user interface, and the rigor afforded by formalizing and proving properties of a system in a theorem prover. The ability to model a system using the graphical editor substantially speeds the process and increases the confidence level, when compared to describing the system as expressions in a language. The ability to visualize the graph is an early aid to catching simple but important mistakes. The ability to explore sample executions through animation helps the user to deepen his understanding of state machine and to do a limited degree of testing. The highest degree of assurance is provided by stating and proving the needed properties of the system using the prover.

There are limitations to the use of a graphical editor once the system begins to get large. It is difficult to comprehend, let alone draw, a system that has in excess of one hundred nodes. Programming languages have used modules as a technique for controlling the complexity of systems. The purpose of the work discussed in this paper is to extend DOVE with the ability to build systems by composing them from simpler component machines.

2 Overview of DOVE

DOVE is primarily a tool for producing high assurance system designs. It provides tools for constructing, presenting and reasoning about formal design-models. DOVE is built in layers with a graphical user interface that is used for constructing and examining the design-models, and an underlying layer using the theorem prover Isabelle. The graphical interface of DOVE is written using Tcl/Tk script language. Isabelle is built in the functional programming language ML, and the proof is carried out by Isabelle.

* This work was supported by ARO Contract Number: DAAD-19-01-1-0473.

DOVE uses a state-machine mechanism to model the specification of system behavior. A state machine in DOVE introduces the notion of a memory at each state, which is updated by each consecutive transition which describes how to evolve between states. The state machine graph consisted of nodes and edges which represent states and transitions. There must be at least one node in the state machine and exactly one node to be defined as the initial state. Each transition has three parts: Let, Guard and Act. The Let part is used to simplify the other two parts of the transition definition. The transition is only performed if the guard is satisfied in the correct memory. The Act, referring to action, defines how the memory is changed by the transition (which only can occur when the transition is performed).

In addition to the visual inspection that the graphical interfaces allows, DOVE provides two other mechanisms for analyzing system designs, namely animations and verification. Animation in DOVE begins by setting initial values for the heap variables (i.e. setting the initial memory), and then is carried out by clicking edges of the state machine graph and calculating new values for the heap variables in accordance with the the corresponding transition definition. This symbolic feature provides a useful way to check whether all variables are updated as expected and whether the transition, which is protected by the guard definition, is performed correctly. However, the animation only gives a simple assurance of correctness of the design of the state machine. A higher level of assurance can be gained by proving whether the design satisfies the given requirements.

Verification in DOVE provides powerful facilities to express properties and to prove the system satisfies system requirements. The system requirements must be translated from informal English into a particular version of linear temporal logic supported by DOVE. DOVE then provides a collection of proof rules and tactics specialized for proving these linear temporal logic properties.

We have applied the DOVE tool to some medium-sized critical systems. The precise details of those applications are not relevant to this paper and are not included here. However, we will include a brief example motivated by our application as an illustration of some of the features of DOVE, and the limitation we wish to address here. The example system is intended to monitor the behavior of another device. The example system consists of two components: a component for monitoring whether the device is plugged in and receiving adequate power, and a component for monitoring when the device is adequately powered whether it is producing values within an acceptable range.

Figure 1 shows a screen snapshot of the DOVE canvas for the PlugIn Monitor component of the system. The gridded canvas is the DOVE state machine window which is used for designing the machine. The three nodes representing the three states in the PlugIn Monitor model are Wait, CheckPlugin and CheckUnplug. The edges with appropriate labels are transitions between these states. Several variables are needed. The heap variable PluggedIn represents whether the machine is plugged in. the input variable Volt is supplied by the environment and is monitored to trace when the device is properly plugged in. Finally, an initial state Wait should be defined in which the machine is unplugged.

The system checks whether the device is plugged in before going from Wait to CheckPlugIn mode. We have variable Volt as the guard for the three transitions: PlugIn, Unplug and RePlugin. At each transition, if the guard conditions are meet, the corresponding transition will be taken, and the variables will be updated. In the initial state, if the device is plugged in and receiving a voltage greater than 10 volts, the transition Plugin will be taken and PluggedIn will be set to true. The plug monitor will stay in the CheckPlugin state unless the voltage drops below 9 volts. In that case, it will enter CheckUnplug state and PluggedIn will be updated to false. Once the device is replugged in and receiving more than 10 volts, it will reenter the CheckPlugin state. The monitor it will keep running in this loop indefinitely. Here, the PlugIn Monitor is correctly and clearly modeled in DOVE.

However, the PlugIn Monitor is just a simple example of modeling a system. Life is not always so easy. When dealing with a bigger project in which some models with interaction with each other, some problem comes up. The ValueOk monitor is a component in which the variable ValueOk shows the status of the value variable. The state machine of Value Monitor is showed as Figure 2. The three states Wait, CheckValueOk and CheckValueFault are defined in the Value Monitor state machine. Six transitions connect these states and update variables if the guard of the transition is satisfied.

In the initial state of Wait, once the variable PluggedIn becomes true, the variable ValueOk will be set to true. The device will enter CheckValueOk state. This can happen in one of two ways. When the system being monitored first starts up, the PlugIn Monitor and the Value Monitor synchronize on beginning to monitor

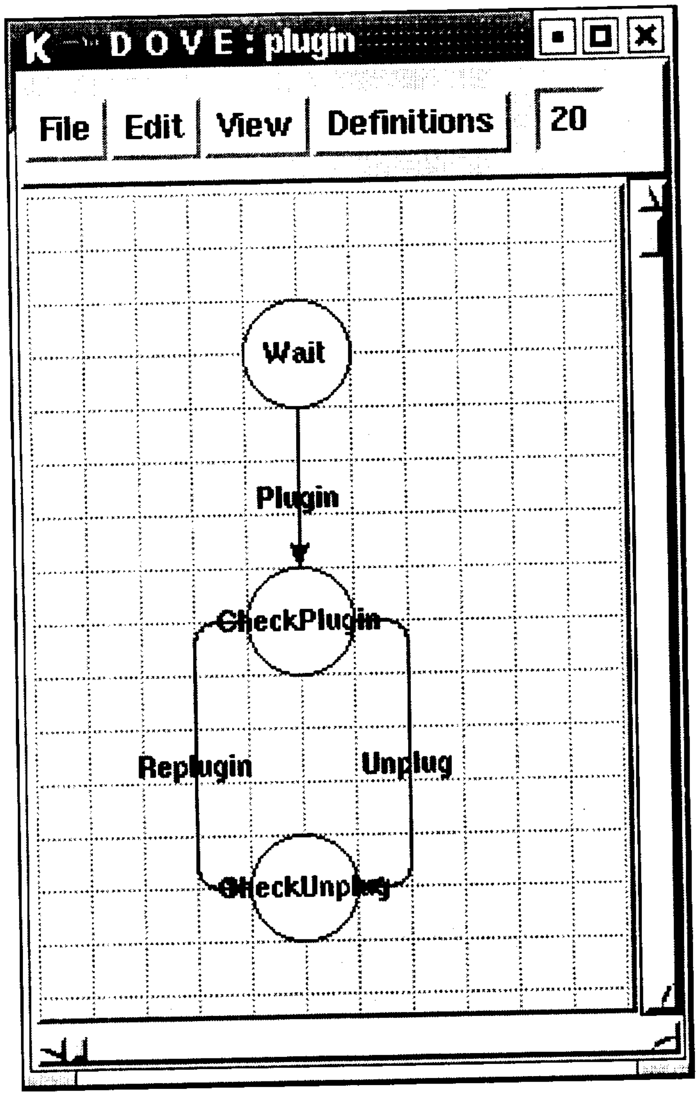


Fig. 1. A simple plug monitor in DOVE

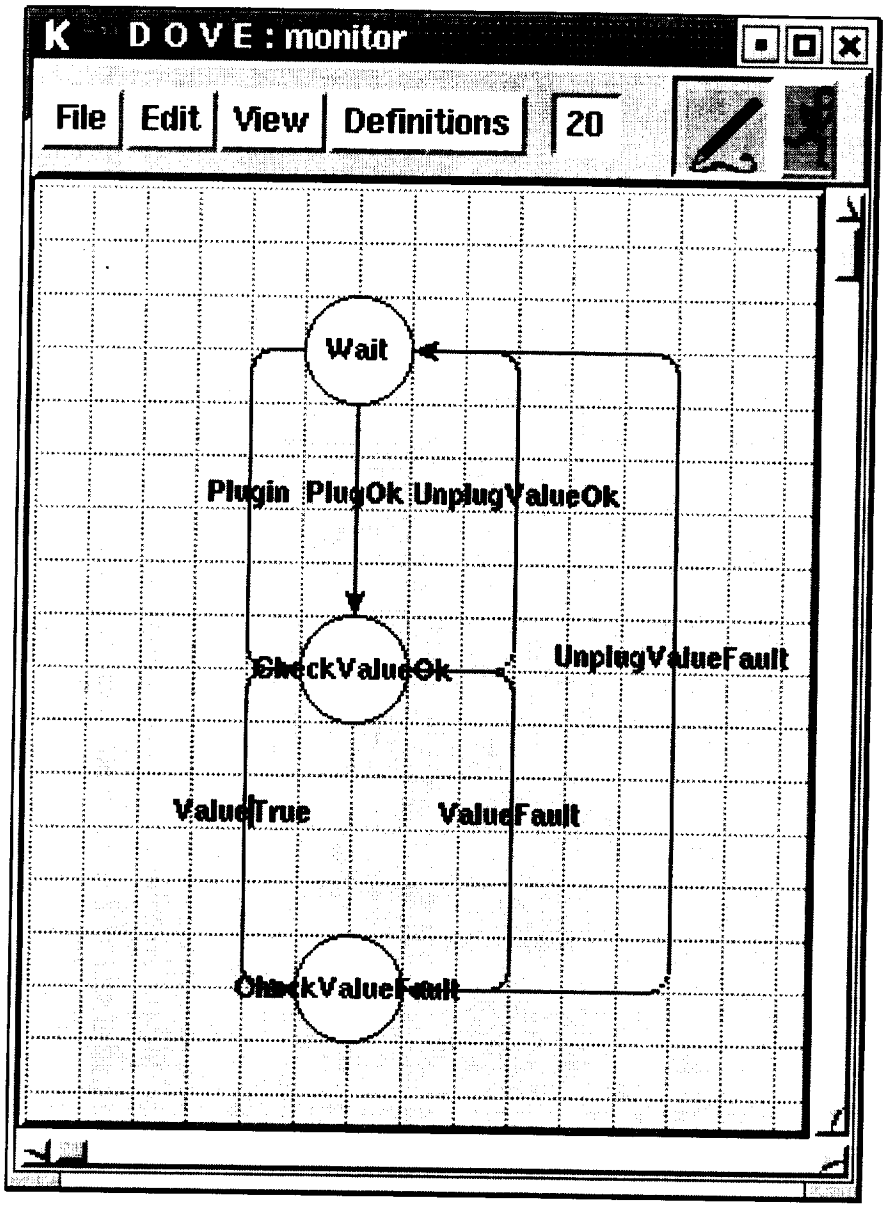


Fig. 2. A monitor for checking values in DOVE

their states. Thereafter, if the power drops below a certain threshold, then the Value Monitor returns to its Wait state, and reenters CheckValueOk when it detects that the PlugIn Monitor has determined that the power has returned to an acceptable level. Once in the ChecValueOk state, if the input variable Test is shown to be below 5, ValueOk is set to false, the device will enter the CheckValueFault state. If variable Test is set back to greater than 5, ValueOk is set back to true, and the CheckValueOk state will be reentered. In both CheckValueOk and CheckValueFault states, if the device is unplugged, the device will go back to initial Wait state.

Between these two models, the ValueOk Monitor uses the PluggedIn variable, which is written by the PlugIn monitor, as an input variable. Unfortunately, with the current DOVE tools, these two interactive components could not be composed into one single model. In order to conquer this, we need to extend DOVE with product automata.

3 Formal Definitions of Automata and Products

In this section we will give a formal definition in higher-order logic of the type of finite state automata used in DOVE, their semantics of execution, and how we extend this with product automata.

3.1 Finite State Machines

Informally, a finite state machine is a tuple of a set of states, a set of labeled transitions, and an initial state. In DOVE, the states are augmented with memory when executed. A transition is a directed edge between a pair of states coupled with a guarded action to be committed when that transition is executed. The transition may be executed only in the case that the guard holds in the memory of the originating state of the transition, and in which case the action yields the memory of the terminating state. Memory is an association of values to variables. The guards are expressed as propositions over the variables in the memory, and the actions are expressed as assignments of values (given as expressions over the memory's variable) to those variables.

This notion of finite state machine (or automaton) is similar to those discussed in the literature, and a typical example can be found in Chapter 4 of [4]. One way in which DOVE extends this notion is by segregating the variables into two categories, in which DOVE are referred to as input variables and heap variables. Input variables are read-only in that no transition may alter their values. Their values are considered to be supplied by the environment. As such, when defining an execution, we must assume that their values may change at any point during a sequence of transitions. While this is manifest in the proof rules in Isabelle for proving temporal formulae for state machines defined in DOVE, it is a subtle point which complicates the definition of an execution and warrants highlighting.

When a user defines a finite state machine in DOVE, they do so using a graphical user interface. This is used to generate a description in Isabelle of the finite state machine and properties that the user wishes to prove. This description of the finite state machines in Isabelle is a shallow embedding in the sense that the variables of the finite state machine are modeled as variables, as opposed to introducing a separate syntax for variables (in the form of a distinct type of variables). Such a light-weight embedding is advantageous when the goal exclusively is proving properties in the model. However, it limits the ability to express meta-properties in the logic, such as stating what a finite state machine is, or what the product of two finite state machines is. Therefore, in this section, we will adopt a deeper embedding. The definition we will give has been rendered in higher-order logic. However, as in the informal description above, it is desirable to express things using set-theoretic notation. In all formal definitions below, such set-theoretic notations should be interpreted as using a standard rendering of naive set theory in higher-order logic, such as one given by sets as predicates.

In attempting to formally define what a finite state machine is, we have to decide how to represent the writable variables versus the read-only variables. Our ultimate goal is to define a product for composing automata, and in such a composition variables which may be read-only in one component may need to be writable in some other. Therefore, we will represent these two classes of variables as disjoint subsets of a single type of variables. For our purposes, the precise type used for representing variables does not matter, so we will use a type variable for this, allowing it latter to be specialized to integers or strings or perhaps some other complex structure. Having made this choice, we will need to be able to express the requirement on transitions that they only involve the variables associated with the particular finite state machine. We will capture this notion of restricted dependence by the following definitions:

$$\mathsf{same\_on}\ dom\ f\ g\ = \forall\ x.x \in dom \Rightarrow (f\ x = g\ x)$$

That is, two functions are the same on a given domain if they have the same values on all elements of that domain.

$$f\ \mathsf{only\_depends\_on}\ s = \forall\ m_1\ m_2.\ \mathsf{same\_on}\ s\ m_1\ m_2\ \Rightarrow (f\ m_1 = f\ m_2)$$

A function on functions only depends on a subset s if it always returns the same value when applied to functions that are the same on s. The motivation for this definition is that our memories are functions assigning values to variables, but the guards and actions are only allowed to depend on that part of the memory that corresponds to the writable and read-only variables.

A transition is well-formed with respect to a set of writable variables and a set of read-only variables provided that the guard and actions depend only on the union of the writable and the read-only variables, and the action does not assign any new values to the non-writable variables.

$$\begin{array}{l}\mathsf{is\_transition}\ (state_1, state_2, guard, action)\ writable\_vars\ read\_only\_vars =\\ \quad guard\ \mathsf{only\_depends\_on}\ (writable\_vars \cup read\_only\_vars)\ \wedge\\ \quad action\ \mathsf{only\_depends\_on}\ (writable\_vars \cup read\_only\_vars)\ \wedge\\ \quad \forall\ memory\ var.\ (\neg(var \in writable\_vars)) \Rightarrow\\ \quad\quad (action\ memory\ var = memory\ var)\end{array}$$

We are now in a position to give a formal definition of a finite state machine:

$$\begin{array}{l}\mathsf{is\_fsm}\ (states, labeled\_transitions, writable\_vars, read\_only\_vars,\\ \qquad initial\_state, initial\_condition) =\\ \quad (writable\_vars \cap read\_only\_vars = \phi)\ \wedge\\ \quad (\forall\ ((s_1, s_2, g, a), l) \in labeled\_transitions.\\ \qquad \mathsf{is\_transition}(s_1, s_2, g, a)\ writable\_vars\ read\_only\_vars\ \wedge\\ \qquad s_1 \in states\ \wedge\ s_2 \in states)\ \wedge\\ \quad (\forall\ (t_1, l_1) \in labeled\_transitions.\\ \qquad (\forall\ (t_2, l_2) \in labeled\_transitions.(l_1 = l_2) \Rightarrow (t_1 = t_2)))\\ \quad initial\_state \in states\ \wedge\\ \quad initial\_condition\ \mathsf{only\_depends\_on}\ writable\_vars\end{array}$$

A tuple of states, labeled transitions, writable variables, read-only variables, initial state, and initial condition is a finite state machine if

- the writable variables and the read-only variables are disjoint,
- the transitions are well-formed with respect to the writable and read-only variables,
- the start and end states of each transition are among the states of the machine,
- each transition label occurs at most once
- the initial state is one of the states of the machine, and
- the initial condition only depends on the writable variables.

3.2 Executions

Up to now we have defined what it means to *be* a finite state machine; we have in effect described its syntax. We are still left with describing how to *execute* a finite state machine; that is, we are left with describing its semantics. The semantics of a finite state machine is the set of all its executions. So what is an execution? Informally, it is a sequence of moves through the finite state machine starting from a memory that satisfies the initial condition of the state machine, and then follows consecutive transitions. More formally, an execution is a pair of initial memories and a sequence of pairs of labeled transitions and resulting memories, where the start state of each transition is the end state of the previous transition and each transition is enabled by the previous memory. However, this is not a complete description. We need to be more precise about what we mean by "resulting memories" and "enabled by the previous memory".

Dove is only capable of dealing with properties that are provable in finite time (safety properties), so we will use lists for sequences. It wold not be fundamentally different if we extended to both finite and infinite sequences.

For the sake of readability, we shall make a couple of short definitions.

$$\begin{array}{l}(\mathsf{last\_state}\ \mathit{initial\_state}\ [\,] = \mathit{initial\_state})\ \wedge\\ (\mathsf{last\_state}\ \mathit{initial\_state}\ (\mathsf{CONS}\ (((s_1, s_2, g, a), l), \mathit{memory}) :: \mathit{seq}) = s_2)\end{array}$$

The last state in a list of pairs of labeled transitions and memories is the initial state if the list is empty, and otherwise is the end state of the transition at the head of the list.

$$\begin{array}{l}(\mathsf{last\_memory}\ \mathit{initial\_memory}\ [\,] = \mathit{initial\_memory})\ \wedge\\ (\mathsf{last\_state}\ \mathit{initial\_memory}\ (\mathsf{CONS}\ (((s_1, s_2, g, a), l), \mathit{memory})\mathit{seq}) = \mathit{memory})\end{array}$$

The last memory in a list of pairs of labeled transitions and memories is the initial memory (for the intended execution) if the list is empty, and otherwise is the memory at the head of the list.

An execution in a finite state machine starting from an initial memory is a list of pairs of labeled transitions from the finite state machine and memories such that

- either the list is empty or
  - the tail of the list is an execution
  - the last state of the tail of the execution is the start state of the next transition
  - the guard is enabled in some memory that is the same as the previous end memory on the writable variables (we allow the read-only variables change) and in that memory we execute the action to acquire the new memory.

$$\begin{array}{l}\mathsf{is\_execution}\ (\mathit{states}, \mathit{transitions}, \mathit{writable\_vars}, \mathit{read\_only\_vars},\\ \qquad\qquad \mathit{initial\_state}, \mathit{initial\_condition})\ \mathit{initial\_memory}\ \mathit{config\_list} =\\ \quad \mathsf{is\_fsm}(\mathit{states}, \mathit{transitions}, \mathit{writable\_vars}, \mathit{read\_only\_vars},\\ \qquad \mathit{initial\_state}, \mathit{initial\_condition})\ \wedge\\ \quad \mathit{initial\_condition}\ \mathit{initial\_memory} \wedge\\ \quad ((\mathit{config\_list} = [\,])\ \vee\\ \quad\ (\exists\, s_1\ s_2\ \mathit{guard}\ \mathit{action}\ l\ \mathit{memory}\ \mathit{tail\_seq}.\\ \qquad (\mathit{config\_list} = (\mathsf{CONS}\ (((s_1, s_2, \mathit{guard}, \mathit{action}), l), \mathit{memory})\mathit{tail\_seq}))\ \wedge\\ \qquad \mathsf{is\_execution}\ \mathit{tail\_seq}\ \wedge\\ \qquad ((s_1, s_2, \mathit{guard}, \mathit{action}), l) \in \mathit{transitions}\ \wedge\\ \qquad (\mathsf{last\_state}\ \mathit{initial\_state}\ \mathit{tail\_seq} = s_2)\ \wedge\\ \qquad (\exists\, \mathit{mem}.\ \mathsf{same\_on}\ \mathit{writable\_vars}\ \mathit{mem}\\ \qquad\qquad (\mathsf{last\_memory}\ \mathit{initial\_state}\ \mathit{initial\_memory}\ \mathit{tail\_seq})\ \wedge\\ \qquad \mathit{guard}\ \mathit{mem}\ \wedge\\ \qquad \mathit{action}\ \mathit{mem} = \mathit{memory}\end{array}$$

We do not intend to go into the details of the particular linear temporal logic used in DOVE in this paper, but briefly a finite state machine is said to satisfy a given linear temporal logic formulae provided every sequence of memories derived from the executions of the finite state machine satisfies the formulae.

3.3 Product Automata

Having defined the syntax and semantics of finite state machines, we are in a position to give the definition of the product of two finite state machines. Using the labels on the transitions, our product will allow synchronization of transitions having the same label. The states of the product is the subset of the product of the states that occurs in the set of transitions of the product (together with the product of the two initial states, if it is not already there). The transitions are effectively the merging of those transitions from the two automata that have the same label, unioned with the remaining transitions lifted to the product states. We have to take a little care to generate distinct new labels for the transitions. The writable variables are just the union of each set of writable variables. The readable variables are the union of each set of readable variables, minus any that are in the union of the writable variables. The product automaton will only be defined in the case where the writable variables of each are disjoint. The variables that are in the intersection of the union of the writable variables and the union of the readable variables are those that are communicating values between the automata. The initial state is just the product of the two origin initial states, and the initial condition is the intersection of the original initial conditions.

Let the states of a transition be its start state and its ending state.

$$\mathsf{statesof}\ ((state_1, state_2, guard, action), label) = \{state_1, state_2\}$$

The product is defined as

```
$wvars_1 \cap wvars_2 = \emptyset \Longrightarrow$
$\mathsf{fsm\_prod}\ (states_1, transitions_1, wvars_1, rvars_1, init\_state_1, init\_cond_1)$
        $(states_2, transitions_2, wvars_2, rvars_2, init\_state_2, init\_cond_2)$ $=$
let $prod\_trans$ $=$
  $\{(((s_1, s_2), (s'_1, s'_2), (\lambda m.g_1 m \wedge g_2 m), a_1 \circ a_2), (l, \mathsf{NONE}, \mathsf{NONE}))\ |$
          $((s_1, s'_1, g_1, a_1), l) \in transitions_1\ \wedge$
          $((s_2, s'_2, g_2, a_2), l) \in transitions_2\} \cup$
  $\{(((s_1, s_2), (s'_1, s_2), g, a), (l, \mathsf{NONE}, \mathsf{SOME}\ s_2))\ |$
          $(s_1, s'_1, g, a) \in transitions_1\ \wedge\ \neg\exists\ t.(t, l) \in transitions_2\} \cup$
  $\{(((s_1, s_2), (s_1, s'_2), g, a), (l, \mathsf{SOME}\ s_1, \mathsf{NONE})\ |$
          $(s_2, s'_2, g, a) \in transitions_2\ \wedge\ \neg\exists\ t.(t, l) \in transitions_1\}$
and
  $prod\_states = \{(init\_state_1, init\_state_2)\} \cup \bigcup_{t \in prod\_trans} \mathsf{statesof}\ t$
in
$(prod\_states,\ prod\_trans,\ wvars_1 \cup wvars_2,$
 $(rvars_1 \cup rvars_2) - (wvars_1 \cup wvars_2),\ (init\_state_1, init\_state_2),$
$\lambda\ m.init\_cond_1\ m\ \wedge\ init\_cond_2\ m)$
```

It follows from this definition that the product of two finite state machines is again a finite state machine, provided their writable variables are disjoint. Note that if the writable variables the first automaton are disjoint from the second automaton, then $a_1 \circ a_2 = a_2 \circ a_1$ (for all a_1 and a_2 in the definition of the transitions in the product automaton above). Therefore, the product of two automata in one order is isomorphic to the product in the other order.

Given an execution sequence, we can project that execution sequence to an execution sequences of of each of the component automata.

```
$(proj_1 (states_1, trans_1, wvars_1, rvars_1, init\_state_1, init\_cond_1)\ [] = []) \wedge$
$(proj_1 (states_1, trans_1, wvars_1, rvars_1, init\_state_1, init\_cond_1)$
     $(\mathsf{CONS}\ ((t, l), mem)\ tail\_seq) =$
  if $\exists\ t'.\ (t', l) \in trans_1$
    then $\mathsf{CONS}\ (((\mathsf{choose}\ t'.\ (t', l) \in trans_1), l), mem)\ (proj_1\ tail\_seq)$
  else $proj_1\ tail\_seq$
```

We can prove that if a given initial memory and sequence of transition-memory pairs is an execution of the product automaton, then the same initial memory together with the projection of that sequence is an execution of the corresponding component automaton. Therefore, for every sequence of memories derived from an execution in the product automaton, there exists an almost identical sequence of memories derivable from a sequence in the component automaton. (The original sequence may have additional memories that are the same as their immediate predecessors in the sequence on the writable variables of the component automaton.) Therefore, for an appropriate class of temporal logic formulae (those that only involve the writable variables of the component automaton, and are "stuttering" invariant), if a formulae holds of the component automaton, it automatically also holds of the product automaton. It is our hope in future work on this system to be able to incorporate this into DOVE.

4 Extending Dove with Products

In the previous section we described the mathematics of the product of two automata. In this section we will discuss our method of implementing the construction of product automata as an extension to DOVE. Our current approach is to add an external tool that can parse files produced by DOVE, analyze the contents of those files to determine the details of the component automata to be composed, construct the product automaton, determine layout information for it, and finally output all this information into a new file that can be input into DOVE.

In the course of a DOVE session, various local files are created, such as an `smg` file, a `thy` file, an `nw` file, etc. The `smg` file, which stands for "state machine graph" file (for example, `plugin.smg`), contains all of the information required to describe the finite state machine. This file includes not only the construction and layout information about the state machine graph, but also the information to define variables, state conditions and transitions between states.

An `smg` is a sequence of lines, each beginning with keyword, followed by data relevant to the item being added. Firstly, the `smg` file gives some preferences for display of the state machine. The global variable `gridOn` tells us the canvas is gridded by being set it to 1, and not gridded by being set it to 0. The variable `edit_SetGridSize` says the size of the grid.

The nodes in the `smg` file are defined using the keyword `file_RestoreNode` follow by the node number, node coordinates and node name. For example, in the plugin state machine graph file, we define the Wait state by

```
file_RestoreNode 0 {20.0 10.0} Wait
```

The node number of Wait is 0 and it is located at (20.0, 10.0). The edges in the plugin `smg` file are created by the keyword `file_RestoreEdge` followed by edge number, the number of the starting node, the number of the ending node, their directions, some coordinates it travels through, and the location of the label and its name. For example, the edge `Plugin` in the plugin `smg` file is defined as follows:

```
file_RestoreEdge 0 0 north 1 south {{20.0 13.0}} {{20.0 11.0} {20.0
12.0} {20.0 13.0} {20.0 14.0} {20.0 15.0}} {20.0 13.0} Plugin
```

In this example, its edge number is 0, it comes out from north of the node 1 and goes into the south of node 1, its label, `Plugin`, is at (20.0, 13.0), and it travels through the path of [(20.0, 11.0), (20.0, 12.0), (20.0, 13.0), (20.0, 14.0), (20.0, 15.0)].

The `smg` file gives two kinds of variables, heap variables and input variables. The heap variables are defined using the keyword `dvd_def`. It is followed by information about their names, types, status and some comments on them. Also we define input variables by `div_defs` followed by the same information as the heap variables.

As for the definition of the transitions, the `smg` file use `dtr_defs`. It gives a list of all the transitions followed by details of individual transitions. These details include the comments, status and the content of the transitions. The content of transition has guard and act definitions in it.

Also it should have an initial state which is defined by the variable `di_startStatea` value. The initial condition is given by setting the variable `di_predicate`. And we can add some comments on the initial state by `di_description`.

In addition, the `smg` file contains some optional information about the finite state machine. For example, if the state machine has been checked and there is no syntax errors, the variable `dchk_smgChecked` is set to be 0, otherwise it equals 1.

From all the information above, we already know enough information to construct the state machine. Any modifications of the `smg` file will directly change the state machine in DOVE. By creating a new `smg` file, we can generate a new finite state machine without starting up the DOVE. We can construct the finite state machine to which is the composition of more than one component in one model without the need to interact with DOVE.

Using the above information, we must parse the smg files of component automata to extract information to reconstruct the automata. Once the automata have been reconstructed, we must build the product automaton. For this we follow quite closely the mathematical description given in the previous section. SML data types and functions may be used to compute the constructions previously given as mathematical formulae. Once the product has been constructed, we still need to generate layout information before we can generate a smg file to add the product automaton to DOVE.

In DOVE, layout information is generated from interactions with the user. The user places nodes at various locations on the drawing canvas and draws edges between the various nodes, indicating curvature by the path of the mouse. The layouts may be altered clicking and dragging the various entities to be changed. DOVE does some work to generate a decent presentation of the graph, but the basic layout information comes from the user. When we automatically generate the product automata, we must also automatically generate some positioning for the components; to make the user generate this information would be almost tantamount to making the user create the product in the first place. To generate this information, we make use of the graph visualization tool *dot* ([3]). Dot is applied to a file that lists the nodes and edges of a directed graph, together with any desired labellings of the nodes and edges, and the desired shape (and color) of the nodes. The following is an example of a part of an input file for dot for the product of the two automata given in Section 2:

```
digraph G {
n8 [label = "CheckPlugin_Wait", shape = circle]
...
n4 [label = "CheckPlugin_CheckValueOK", shape = circle]
...
n0 [label = "Wait_Wait", shape = circle]
n8 -> n4 [label = PlugOK_at1_CheckPlugin]
...
n0 -> n4 [label = "Plugin"]
...
}
```

Dot then generates a layout for the graph and out puts it in one of several formats, including gif and postscript, for example. The mode we used is an expanded form of the same language used for input, where layout coordinates have been added. The above graph description is translated to:

```
digraph G {
node [label="\N"];
graph [bb="0,0,2211,516"];
n8 [label=n8, shape=circle,

n8 [label=CheckPlugin_Wait, shape=circle, height="0.56",
pos="1128,488", width="0.56"];
...
n4 [label=CheckPlugin_CheckValueOK, shape=circle,
height="0.56", pos="1028,212", width="0.56"];
...
n0 [label=Wait_Wait, shape=circle, height="0.56",
pos="1709,304", width="0.56"];
n8 -> n4 [label=PlugOK_at1_CheckPlugin, pos="e,1048,214
1139,471 1165,429 1223,318 1171,250 1156,231 1092,221
1055,215", lp="1270,350"];
...
n0 -> n4 [label=Plugin, pos="e,1048,213 1692,293 1685,289
1679,285 1675,284 1557,244 1168,219 1057,213", lp="1633,258"];
...
}
```

For each node, the size (height and width) of the circle, and the position of its center is added. Each edge is extended with path information, consisting of the position and direction of the terminating arrowhead follow by a sequence coordinates that the edge will pass through, and with the coordinates of the left edge of the label.

We must parse the information returned from dot and combine it with the non-graphical information for the product automaton (such as the guards and actions for each transition). Also, the graphical information produced by dot is not completely suitable for directly inputting into an smg file. We need to perform scaling, and better layouts seem to be given by thinning the points for layout of the transitions. Once we adjust the information from dot and combine it with the non-graphical information, we can finally produce an smg file that describes the product automaton to DOVE. Once this file exists, the user can start up DOVE with it, and proceed to state and prove properties about it.

5 Future Work

The work described in this paper outlines a way to build the interactive components into one finite state machine by extending DOVE with product automata. By using the information we get from parsing the smg file in DOVE, we can create a new state machine graph file by hand without disturbing DOVE. Programming to perform all of the steps indicated in Section 4 is not yet finished and tested. We anticipate having a completed prototype by the time of the conference.

Once the product automata is built, we also need to test its correctness and feasibility. We begin this project because we were attempting to use DOVE to reason about a medium-sized real-world safety-critical system. This system could be naturally decomposed into a hierarchy of subsystems communicating through limited intefaces of input and output variables. In attempting to use DOVE, we found ourselves attempting to compose these subsystems by hand. With the completion of this tool, we will return to this example and use DOVE to describe this hierarchy and complete the task of proving the required properties.

As described in this paper, we are adding a tool to DOVE that will allow for the automatic construction of product automata from component automata. However, there is more that we desire. At the end of Section 3, we indicated that the mathematical theory underlying the product automata should allow us to automatically translate properties that hold of an individual component automaton to corresponding properties of the product automaton. DOVE should be extended to support such a feature. The user should be able to reason about the various components and then have those results automatically carried over to the product when the product is formed or its theory is subsequently updated. To support this with the full rigour currently available in DOVE, we would need to be able to prove in Isabelle that the product that we have externally created is indeed the product as mathematically defined. To be able to prove such a fact requires a deeper embedding of finite state machines in Isabelle than is currently used. Therefore, adding this extension would require a siginficant reworking of the foundations of DOVE. It is our opinion that the benefits would merit such an effort.

References

1. Tony Cant, Katherine Eastaughffe, Jim Grundy, Jim McCarthy, Brendan Mahony, Maris Ozolos, Tim Anstee, Moira Clarke, Geoff Tench, Helen Daniel, Tony Dekker, Mark Klink, Chuchang Liu, John Yesberg. DOVE User Manual. December, 2000.
2. M. A. Ozols, K. A. Eastaughffe, A. Cant, S. Collignon. DOVE: A Tool for Design Modelling and Verification in Safety Critical Systems, in *Proceedings of the 16th International System Safety Conference*, Seattle, US September 1998.
3. Eleftherios Koutsofios, Stephen North. Drawing graphs with *dot*. AT&T Bell Laboratories Technical Report, 1996.
4. Doron A. Peled. *Software Reliablity Methods*. Texts in Computer Science. Springer-Verlag, New York, 2001.
5. Lawrence C. Paulson. *The Isabelle Reference Manual*. 2000.

A Higher-Order System for Representing Metabolic Pathways*

Sara Kalvala

Department of Computer Science
University of Warwick, UK
Sara.Kalvala@warwick.ac.uk

Abstract. This paper introduces the topic of metabolic pathways and explores it as a subject for study by the theorem-proving community. A description of the issues involved is provided, as well as a justification of why a logic-based description of these pathways might complement the current progress in the area of Bioinformatics.

1 Introduction

Bioinformatics is the application of computational methods in the understanding of biological systems. Typically, it involves analysing information stored in large databases; the information itself is obtained from experiments.

The processes of gene expression and protein function are schematized in Figure 1. Bioinformatic research on these processes has given rise to three sub-areas:

genomics is the deciphering of the code contained in the DNA, that is knowing what the actual strings are and which genes exist; understanding how the code of the DNA is actually expressed

proteomics is concerned with understanding the functioning of proteins, which structurally are the products of DNA translation and functionally are the active agents of life, whether as enzymes or channels or any other way.

metabolomics studies the biochemical processes that occur within cells, and the complexities of control that make living organisms.

Each of these topics of research has given rise to a large variety of formalisms developed by often competing groups. An important issue is finding the right abstraction which allows the different tools to work. Each of these systems has its own language. [1] discusses the fragmentation of bioinformatics protocols, technologies and standards, that together created a landscape of confusing and poorly integrated web sites and other services, and suggests that the solution may be a formal model to unify these languages.

The success of research in genomics can be attributed partly to the use of a very simple abstraction, namely the four letters A,T,C, and G organised into substrings (the genes), which are then strung together to form chromosomes. Similarly, proteins are formed from a larger alphabet of amino-acids. A paper by Giegerich, Hinze and Kurtz [3] presents a small model of DNA/proteins in Haskell (Figure 2). Note that this description of the process of DNA transcription does not scale up to the

However, when we come to proteomics, we start encountering difficulties, as there is no simple representation for the *shape* of proteins, and it is still not possible to predict the shape of a protein from knowing its chemical structure.

Metabolism can be seen as a complex network of reactions depending on the interaction between many different kinds of molecules. The primary actors are *enzymes* - proteins that facilitate reactions. Here also there is a problem with finding the appropriate abstraction for representing the biological system. Graph theory is usually part of the model, and there are different programs for displaying graphs in a pleasant fashion on a 2D screen. However, the additions that need to be made to the graphs are:

* Research done partially while author was a Visiting Fellow at the Australian National University.
[1] See http://www.oreillynet.com/pub/a/network/2002/01/29/bioday2.html

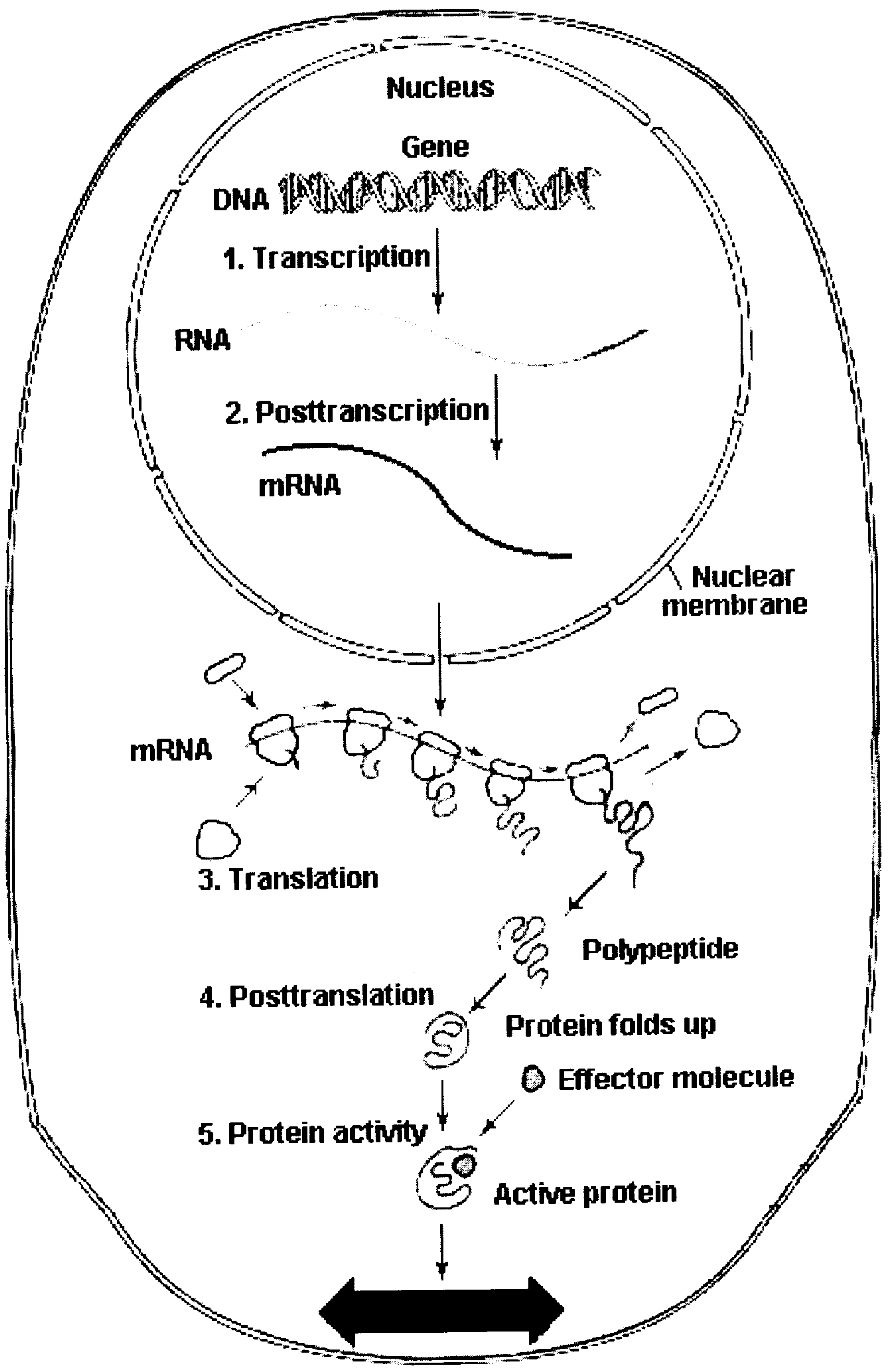


Fig. 1. From DNA to protein function (from `www.people.virginia.edu/ rjh9u/trtrpict.html`)

```
data Nucleotide = A | C | G | T
data AminoAcid = Asn | Lys | ... -- and so forth (aminoacids)
type DNA = [Nucleotide]
type Protein = [AminoAcid]
type Codon = (Nucleotide, Nucleotide, Nucleotide)

genCode :: Codon -> AminoAcid

genCode (A, A, A) = Lys; genCode (A, A, G) = Lys;
genCode (A, A, C) = Asn -- and so forth

ribosome :: DNA -> Protein
     -- the ribosome always starts at ATG
ribosome (A : T : G : x) = Met : map genCode (triplets x)

triplets [] = []
triplets (a : b : c : x) = (a, b, c) : triplets x

wc_compl A = T; wc_compl T = A;
wc_compl C = G; wc_compl G = C

complSingleStrand [] = []
complSingleStrand (a : x) = wc_compl a : complSingleStrand x

dnaPolymerase x = (x, complSingleStrand x)
```

Fig. 2. Genomics in Haskell

- *classes* of chemicals
- *classes* of reactions
- inexact matching of graphs

The data needs to be stored in a rich formalism. The number of macromolecules, reactions, combinations, etc is huge. The analysis needs to take into account these interrelations between entities. In this paper we explore the use of higher-order logic to represent and manipulate metabolic information.

2 Biochemical reactions and enzymes

A metabolic pathway is a sequence of biochemical reactions. These reactions are rigorously controlled by a complex mechanism, in order to maintain balance within cells. The most direct form of control is through a *catalyst*. Catalysts are not consumed by a reaction, so that once a molecule of the catalyst facilitates a reaction, the products are produced, and the catalyst molecule is released back into the environment, potentially allowing it to facilitate another instance of the reaction. Theoretically, the reactions would still proceed in the absence of the catalyst, but in a slower rate, sometimes close to zero. It is useful for biochemical reactions to depend on catalysis, in order to provide control mechanisms.

Most chemical reactions have a hidden component: the energy involved. In inorganic media energy is mutated into heat, but in living organisms it is stored into complex molecules, which serve as a buffer. Cellular energy management is based around pairs of related molecules, where one contains a substantial amount of energy compared to the other. Typical pairs are ATP and ADP (Adenosine Tri and Di Phosphates), GTP and GDP, and NAD+ and NADH. When energy is required for another reaction, the high energy molecules are used, and replenish the stock of low energy molecules which then are used when more energy is released elsewhere.

There are other sources of more concentrated energy as well, e.g. in the form of lipids, but these take longer to release. This energy is first transferred into ATP/GTP molecules which are then used *in situ*.

Typically one direction of a reaction releases energy while the other consumes, or ***stores*** energy. Most reactions are theoretically reversible, but the difference in energy means that one direction is favoured over another. Furthermore, even if a reaction ultimately releases energy into the medium and is favoured, it often needs a lot of energy to be started. Catalysts often work by reducing the energy required for a reaction to occur, and thus can increase the rate of reaction.

While catalysts can be of one of many types, ***enzymes*** are catalysts primarily made up of proteins. Typically, an enzyme facilitates a reaction by slotting reactant molecules into particular spaces and thus bringing them into close proximity and at the right spatial relation so that the reaction can occur; once this happens the product molecules are released into the environment and new reactants occupy the now vacant spaces close to the protein.

2.1 Classification of enzymes

Enzymes are a large group of proteins; enzyme list from 1992 contains 3196 live entries. With such a large number, it is important to classify them in a meaningful and useful way. The most accepted method of classification of enzymes is provided by their EC number. This is a strictly functional classification, based on 4 numeric fields [1]. The first field specifies the kind of reaction the enzyme catalyzes, while the second one describes the active atoms involved. This is illustrated in Figure 3

1. Oxidoreductases
   1.1. CH-OH
   1.2. OXO
   1.3. CH-CH
   1.4. CH-NH(2)
   1.5. CH-NH
   1.6. NADH/NADPH
   1.7. N compounds
   1.8. Sulphor
   1.9. ...
2. Transferases
   2.1. 1-carbon group
   2.2. ketone residues
   2.3. acyl
   2.4. ...
3. Hydrolases
   ...
4. Lyases
5. Isomerases
6. Ligases

Fig. 3. Classification of enzymes

It should be remembered that this classification is ***functional***, and the information is usually derived from biochemical experiments. But enzymes should also be understood to be essentially proteins, produced by the translation of a gene and then the activation of the protein by taking a particular 3-dimensional shape, which is a consequence of interaction of its particles with each other and with the environment. Correlating a known enzyme function to a particular gene is a matter of intense research in the field of bioinformatics.

As an example, one may look at a particular place in the EC classification: EC 1.1.99 consists of malate dehydrogenases. But there are many malate dehydrogenases, from different organisms. EC 1.1.99.16 represents a variety of proteins, ie NCBI 1788539 on E.Coli and NCBI 2078007 in P. Aeruginosa. A comparison of the sequences reveals that they are of different size and are of different sequence so how do we know they are the same enzyme? An ***ortholog cluster*** is such a group of functionally related enzymes, one from each organism, which can be ***postulated*** to have a common origin. Confirmation of this link can be partially obtained by applying string matching algorithms, but this is inadequate because the relation between differences in the aminoacid structure of proteins and their functionality is not linear. Another approach to correlation of enzymes lies in understanding the pathways in which the enzyme is used. The understanding of these relations is crucial for several of the applications of bioinformatics, such as understanding etiology of diseases and aiding drug design.

3 Understanding metabolic pathways

A pathway consists of a sequence of biochemical reactions. Pathways can be classified into several types: synthesis, degradation, and energy transfe. Each of these pathways is made of reactions that can be classified in a way similar to the enzyme classification of Figure 3. As many of the molecules in a pathway are recycled, quite often pathways are shown as cycles, specially when one of the recycled elements is of larger complexity than some of the products, or when it appears in very few other pathways.

One of the most well known pathways is what is known as Kreb's cycle, or TCA cycle, illustrated in Figure 4. This pathway is shown as a cycle because a Citrate molecule is regenerated each time, which is then used in successive reactions to yield high energy NADH and GTP molecules. The citratemolecule is a product of the reaction of oxaloacetate with acetyl CoA (co-enzyme A) which is the productderived from "burning" of sugar. One may note the presence of several enzymes (indicated by the -ase suffix) which drive this cycle.

Metabolic networks are very robust, due to several factors:

1. failure of an enzyme due to a structural change is not always catastrophic, in the sense that impairment of an enzyme's function can be partial;
2. if a particular enzyme is non-functional, other enzymes which otherwise have only a weak action can be modulated to have further effect, as many enzymes can work on different pathways;
3. there may be several pathway *variants*: similar functions using different reactions and routes, which can continue ellular function in a reasonable manner.

Within a natural environment, a pathway does not occur in isolation, but is part of an elegant, complex system [6], where molecules are resources that are shared between many processes, or which are made available through diffusion or membrane transport mechanisms. Also, innumerable *instances* of reactions from different pathways occur simultaneously and in close proximity, and the balance between the concentrations of all these elements is intricate. The separation into individual pathways, and the concretization into individual molecules, is merely didactic.

Furthermore, one needs to take into consideration the compartmentalization of the environment through permeable membranes. Proper cellular function is tied to the mechanisms for *transport* of molecules — whether through simple diffusion or across a membrane (between the cytoplasm of a cell and its environment or between the cytoplasm and the nucleus or organelles such as the mytochondria) through *channels*. Quite often proteins are involved in these transport mechanisms, and they are subject to control strategies similar to the control of reactions.

3.1 Other biochemical networks

Metabolic pathways are just one type of biochemical networks. Other networks are gene regulatory networks and signal transduction networks, which make sure that the biochemical reactions occur at rates that are beneficial to the organism, depending on extra-cellular factors as well as internal feedback mechanisms.

A simple form of control for enzymatic catalysis is the control of the production of the proteins from DNA. The more proteins are created and activated as enzymes, the higher the rate of reaction. By stopping the production of enzymes, eventually the concentration will bereduced, and the rate of reaction will decrease. However, the process of protein production may be slow, and the rate of natural degradation of proteins might be extremely slow, so other mechanisms have evolved, such as signal transduction networks and hormonal signalling.

As well as having more short-term control of metabolism, organisms also need to perform long-term control. One particular method of control is through the use of *regulons*, which control groups of operons which control gene groupings. Global regulons coordinate regulation of operons in multiple metabolic pathways, other global regulators act through control of DNA spatial configuration. The biochemical logic in genetic

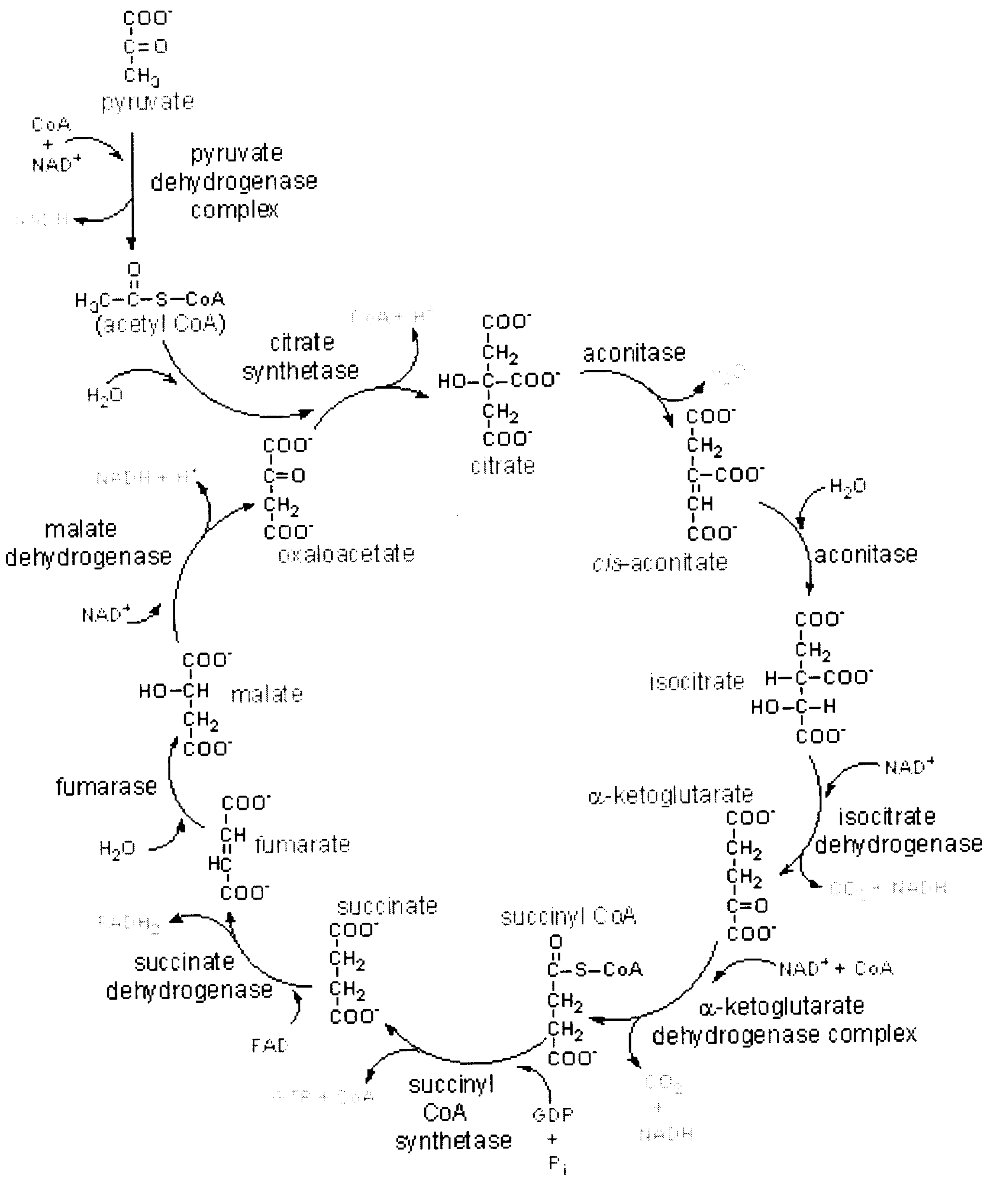


Fig. 4. The TCA cycle (from `http://chemistry.gsu.edu/glactone/PDB/Proteins/Krebs/Krebs.html`)

regulatory circuits provides real-time regulatory control, which can be seen as a branching decision logic, executing stored programs that guide cellular differentiation extending over many cell generations.

4 Representing metabolic pathways

In this work we are interested in developing suitable representations for the kind of information described above. Representing this kind of information in a plain, first-order system doesn't capture the true richness of the domain, because biochemical processes are characterised by their generality and adaptability, as well as the inter-relationships between the many entities. Several open research problems such as discovering evolutionary relatedness, finding alternative pathways, and predicting shape from sequence *depend* on finding these relationships.

Many databases of metabolic information have been developed and are in widespread use by biologists and biochemists. While ideas of object-orientedness and subclassing mechanisms are sometimes exploited, several of the languages and techniques are quite ad-hoc, and many of the features that could be implemented by a good type system are currently processed through extensive explicit programming. We next introduce some of these systems, and then present a vision for a more appropriate representation.

4.1 Existing technology

Biochemical dynamics are sometimes modelled quantitatively, in order to capture the situation. A popular model is the e-Cell [10], which represents a simplified cell in an object oriented language (C++) and allows experiments in which concentrations of molecules can be set and observed at different (virtual) times. However, there is still a lot that can be gleaned about metabolic processes from a purely qualitative model, and this is the approach taken by most of the existing systems.

There are many databases with very impressive amount of information about metabolism, such as KEGG [4], METACYC [5], EMP [9], and UM-BDD [2], amongst others. Often these databases have grown from an attempt to use, organize and share in-house data, and some of the software tools have been developed with these aims in mind.

KEGG (the Kyoto Encyclopedia of Genes and Genomes) is one of the best sources of data. It covers not only metabolic information but also what is possibly the most complete genome database. It is based on the DBGET integrated database, and is also linked with LIGAND (a chemical database for enzyme reaction).

According to Peter Karp, the developer of MetaCyc, KEGG mixes information from different organisms. It also has no information about enzyme inhibitors or subunits, or substrate specificity. MetaCyc contains information about 4218 reactions organized into 445 pathways, obtained second hand from literature, and covers 12 organisms. MetaCyc stores *super-pathways* - groups of pathways linked by common substrates. Pathways are inferred using a module called PathoLogic. Some problems identified with MetaCyc are missing or incorrect information.

WIT is a system for reconstructing metabolic networks based on EMP data [7], and it supports the use of phenotypic data as well as usual biochemical and genomic information. EMP bills itself as the most comprehensive source of biochemical data. The University of Minnesota's UM-BDD focuses on bacterial and archaeal pathways, and the study of *enzymes*. The information is curated from different sources. Amongst all the databases, it is the one where the contribution from different sources, including KEGG and EMP, are acknowledged, and the importance of sharing information is raised.

All these systems seem to work on the premise that metabolic pathways are graphs, ie lists of reaction pairs. The way each of these reactions, and the consituent molecules, is stored and accessed is an engineering issue, but mostly they use a straight-forward database representation. However, because they represent "large, noisy, complex data-sets and knowledge sets" (in the words of Peter Karp), there are bound to be inconsistencies and information of limited certainty. It seems sensible to apply some of the expertise of the theorem-proving community in developing a well-thought out representation for this knowledge.

4.2 Pi-Calculus models of bionetworks

There is substantial work on models of biological networks by Regev, Priami, et al. using Stochastic Pi Calculus [8]. Reactions between molecules are activated by the exchange of signals, and hiding is used to model specificity, and creation of intermediate compounds. Their models are quite detailed, and capture numerical reasoning along with the qualitative description. It is unclear if the numerical results obtained by running the stochastic pi-calculus descriptions correspond to real-world data, and if the semantics of the two systems correspond to each other. Furthermore, the descriptions are given in great detail, but this detail makes it difficult to get a clear picture and reason about the pathways in a structured fashion. And there is scope to explore *families* of reactions, something that would be aided with the use of a richer type-based language, which better capture the general rules and polymorphic nature of metabolism.

5 Use of higher-order formalisms

While the databases that have been described above contain a lot of information about metabolic processes, one aspect they all seem to be lacking is an appropriate *ontology* for the system. The databases take their queue from encyclopaedias, which store information which must then be processed by the scientist who builds ontologies in an informal fashion. There are projects aimed at building ontologies for biosystems (such as the use of Description Logics for building the Tambis Ontology (TaO); these logics aim at improving the understanding of the way information is stored in the databases, rather than looking purely at the biological systems themselves. Typical ontologies use a small number of concepts, such as relations, instances, and axioms.

The use of a logical formalism for describing metabolic data can move the paradigm for computational models of metabolism significantly. Rather than considering a database, where the power exists in the capture of a great amount of details and the existence ofvisualization and access tools, a logical model would offer simplicity and the power to add new information in a simple and consistent manner. Rather than testing a new model for a pathway as a new graph in the database, one could write it as a functional composition of possibly polymorphic functions, and reason about it within the established logical rules and deduce properties such as liveness of the cycle.

Several formalisms which have been developed initially by logicians and later eagerly adapted by computer scientists for the purpose of describing computational systems are particularly suitable for capturing the workings of metabolic pathways:

Temporal logic: a system of inference rules that allows one to reason about the evolution of a system, in terms of eventual outcomes, invariants, entailments, and fairness amongst several processes. The goal is to know the outcomes of a process not by simulating it for a long (abstract) time but by analytical reasoning.

Linear logic: a system of inference rules that allows one to model resources, and propositions that hold at one point and may not hold at the other. One of the main simplifications of traditional inference systems is that once a theorem is said to hold, it needs to be assumed to always hold; this makes it difficult to model transient properties inherent to biological systems. Linear logic solves this issue by allowing some assumptions to be used only a finite number of times, which therefore makes it suitable for representing chemical processes.

Type theory: simple data types do not capture the richness of groupings and dependencies existing in natural systems. Recent interest in object-oriented modelling shows that bioinformaticists are keen on exploring new type systems, but object-orientation itself is more geared towards programmability that description. The interaction between quantification, subtyping, and polymorphism of systems such as $F <:$ allow a richer description of data.

All these approaches can be implemented within Higher-Order Logic, and in fact several implementations in different proof systems already exist. New extensions allow the description of probabilistic algorithms and reasoning about them. In this poster simple models of enzymatic pathways using all three formalisms above will demonstrate the usefulness of these advanced logical systems to bioinformaticists. This is work in progress, and it is expected that a second pass at a logical model that integrates the separate descriptions in the three formalisms above will be developed.

References

1. Nomenclature Committee. *Classification and Nomenclature of Enzyme-Catalysed Reactions*. International Union of Biochemistry and Molecular Biology.
2. Lynda Ellis, Douglas Hershberger, Edward Bryan, and Lawrence Wackett. The University of Minnesota biocatalysis/biodegradation database: emphasizing enzymes. *Nucleic Acids Research*, 29(1), 2001.
3. Robert Giegerich, Ralf Hinze, and Stefan Kurtz. Straight to the heart of computer science via functional programming. In *Proceedings of the Workshop on Functional and Declarative Programming in Education*, 1999.
4. Minoru Kanehisa. Toward pathway engineering: a new database of genetic and molecular pathways. *Science and Technology Japan*, 59, 1996.
5. Peter D. Karp, Monica Riley, Suzanne M. Paley, and Alida Pellegrini-Toole. The MetaCyc database. *Nucleic Acids Research*, 30(1), 2002.
6. Boehringer Mannheim. Biochemical pathways chart. wall chart, 2000.
7. R. Overbeek, N. Larsen, N. Maltsev, G. D. Pusch, and E. Selkov. Metabolic reconstruction using the WIT/WIT2 systems. Technical Report ANL/MCS-P694-0997, Argonne National Labs, 1997.
8. Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80(1), 2001.
9. Evgeni Selkov. Emp database.
10. M. Tomita, K. Hashimoto, K. Takahashi, T. Shimizu, Y. Matsuzaki, F. Miyoshi, K. Saito, S. Tanida, K. Yugi, J.C. Venter, and C Hutchison. E-CELL: Software environment for whole cell simulation. *Bioinformatics*, 15(1), 1999.

Higher-Order Pattern Unification and Proof Irrelevance

Jason Reed

Carnegie Mellon University
Pittsburgh, PA 15213
jcreed@andrew.cmu.edu

Abstract. Extending the type theory of a logical framework with a proof irrelevance modality has several potential advantages, including the ability to represent subset types and invariants for proof compression. Although the extended theory is well-behaved, it is not yet completely clear how to modify the implementation of a logical framework to accommodate proof irrelevance. The unification algorithm in the logical framework Twelf, in particular, works by a process of constraint simplification that depends on the notion of pattern substitution. Adapting this algorithm to work with proof irrelevance requires generalizing the definition of pattern. We propose such a definition, guided by work with proof irrelevance and strictness, and make progress towards proving its correctness.

1 Introduction

1.1 Higher-Order Pattern Unification

Our starting point is an algorithm for higher-order pattern unification using explicit substitutions due to Dowek et al. [3]. The difficulties of variable capture involved in higher-order unification are avoided by a reduction [2] to first-order equational unification in a language with explicit substitutions [1]. This first-order algorithm specializes nicely for a decidable subset of unification problems, the *pattern fragment*. The strategy of constraint simplification for solving general unification problems – trying to solve equations which are in the pattern fragment and postponing constraints produced by those that aren't, which may later be resolved by substitutions arising from other equations – though not complete, has been found to work well in practice.

Informally, *pattern* is a term where all the variables (by which we mean metavariables amenable to substitution) occur above a sequence of distinct deBruijn indices, i.e. bound variables. If all variables have atomic type (and we can easily transform problems to have this property) then this means each variable must occur under a substitution whose range is a set of distinct deBruijn indices, a *pattern substitution*. Pattern substitutions are desirable because they have one-sided inverses, and so are injective. Equations of the form $X[\zeta] = b$ therefore have at most one solution, written $b[\zeta]^{-1}$.

1.2 Proof Irrelevance

Constants are naturally injective; if we have a constant $c : \tau_1 \to \cdots \to \tau_n \to \tau$ of n arguments, the terms $c\ M_1 \cdots M_n$ and $c\ M_1' \cdots M_n'$ are equal if only if $M_i = M_i'$ for all $1 \le i \le n$. There are times when this is not the desired behavior. Sometimes we would like to be able to make certain arguments to a function 'irrelevant' when it comes to deciding equality. This can occur when some arguments to a function are meant to be thought of as witnesses of provability rather than pieces of data whose structure matters. This phenomenon is *proof irrelevance*. Generally, suppose we use the following signature in the dependent type system LF ([4]):

$$\begin{aligned}
&t, u : \mathsf{type}\\
&p : \Pi x{:}t.\,\mathsf{type}\\
&c : \Pi x{:}t.\Pi y{:}(p\ x).u\\
&a : t\\
&b, b' : p\ a.
\end{aligned}$$

Intuitively, we have an object a of type t, and some predicate p on objects of type t. The function c takes two arguments: an argument x of type t, and an argument y of type $p\ x$ (which might be thought of as a proof that p holds of x) and returns an object of type u. As the preceding discussion describes, $c\ a\ b \neq c\ a\ b'$ because not all of c's arguments are the same on both sides of the equation b and b' differ. We would like to declare instead a $d : \Pi x{:}t.\Pi y \div (p\ x).u$, where the $\div$ symbol is supposed to be a promise that the argument y 'doesn't matter' in the output of d. We proceed to discuss some applications of proof irrelevance in more detail.

1.3 Encoding Subset Types

The example above is really a trivial case of the use of proof irrelevance to obtain adequate encodings of *subset types*, which can be useful when representing programming languages and logics in a logical framework. A subset type is a formal version of the common mathematical set-formation syntax $\{x \in X \mid \phi(x)\}$ for some predicate ϕ on X. As a typical example of its usefulness, suppose we wish to represent a lambda calculus which has a 'relevant' binder $\lambda^{\geq 1}$ which requires the variable it binds to occur at least once. We could try doing this by beginning with the usual untyped lambda calculus

$$\begin{aligned}&\mathsf{tm} : \mathsf{type}\\&\mathsf{app} : \mathsf{tm} \to (\mathsf{tm} \to \mathsf{tm})\\&\mathsf{lam} : (\mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm}\end{aligned}$$

and defining a predicate

$$\mathsf{occurs} : (\mathsf{tm} \to \mathsf{tm}) \to \mathsf{type}$$

via the *logic program*

$$\begin{aligned}&\mathsf{occurs\_var} : \mathsf{occurs}(\lambda x.x)\\&\mathsf{occurs\_appl} : \mathsf{occurs}(\lambda x.\,\mathsf{app}\ M_1\ M_2) \leftarrow \mathsf{occurs}(\lambda x.M_1)\\&\mathsf{occurs\_appr} : \mathsf{occurs}(\lambda x.\,\mathsf{app}\ M_1\ M_2) \leftarrow \mathsf{occurs}(\lambda x.M_2)\\&\mathsf{occurs\_lam} : \mathsf{occurs}(\lambda x.\,\mathsf{lam}(M\ x)) \leftarrow (\Pi y{:}\mathsf{tm}\,.\,\mathsf{occurs}(\lambda x.M\ x\ y))\end{aligned}$$

which captures the proposition that an open term uses its free variable, and declaring a constant

$$\mathsf{olam} : \Pi t : (\mathsf{tm} \to \mathsf{tm}).\Pi p : \mathsf{occurs}\, t.\, \mathsf{tm}$$

However, because the binding of the proof p is the usual, "relevant" binding, this encoding is not *adequate*: There are generally multiple LF terms which represent a given object-language term. This is because there are potentially many proofs that a variable occurs bound — in fact there will be one proof per occurrence of the variable. Therefore we want to equate all terms using olam that differ only in which occurrence proof they use. That is, we should declare instead

$$\mathsf{olam} : \Pi t : (\mathsf{tm} \to \mathsf{tm}).\Pi p \div \mathsf{occurs}\, t.\, \mathsf{tm}$$

and the new encoding is adequate. Though the revised olam still requires the existence of a proof p that occurs t holds, it doesn't care *which* proof in a certain sense, which is guaranteed by the type system.

1.4 Subterm Omission in Proof-Carrying Code

In a language like Java, some measure of safety of running code is insured by running the code 'in a sandbox,' inside a trusted virtual machine. Proof-carrying code [5] is another technique which aims to achieve the same (if not greater) safety properties without sacrificing runtime efficiency to emulating a virtual machine. The burden of making a program safe falls instead on the author of the program, the *code producer*. The recipient of a program, the *code consumer* requires a *proof* that the program received satisfies some safety policy, and so the code producer must send with the program a formal certificate of safety which can be mechanically checked by the code consumer to actually prove that the program won't violate the policy.

Unfortunately, these certificates can sometimes be large, even on the order of the size of the size of the program being proved safe. Techniques for reducing proof size are desirable. Although the problem of finding a proof for a given proposition is typically undecidable, a proof may have many subproofs which could be easily and efficiently reconstructed by the code consumer. For instance, as part of a large proof that shows that a program always computes a certain mathematical function correctly, it might be necessary to show some trivial fact, say $3 + 4 = 7$. Now the formal correctness of this program depends on every last detail of the proof being correct, but there is no need to send a proof of $3 + 4 = 7$ across the network there can simply be a blank spot in the proof with an instruction saying, "please check that in fact 3+4=7." The trade-off here is saving network bandwidth by perhaps spending more time reconstructing proofs on the code consumer end.

In practice, proofs are frequently represented as terms in a type theory like LF, and checked with a tool like Twelf ([8]). In this case, the idea of omitting subproofs really means omitting subterms. The question to be addressed is, when can a subterm be safely omitted? Twelf already has facilities for providing sufficient conditions for *termination* of predicates considered as logic programs. When we can show using these tools that searching for a proof of P (which is what is meant by "running the logic program P") always terminates, and we know a proof of P, then we can be sure that the code consumer can also find a proof of P. What we do not know is that the code consumer will find the *same* proof. It may seem like a desirable property of a type system that if we replace a subterm S of a term M with a different subterm S' of the same type, then M is still well-typed, but dependent types systems do not necessarily have this property exactly because of the dependence of types on terms. For example, in the signature

$$
\begin{aligned}
&a, z : \mathsf{type}\\
&b : a \to \mathsf{type}\\
&c : \Pi x{:}a.(b\ x)\\
&d : \Pi x{:}a.(b\ x) \to z\\
&k_1, k_2 : a
\end{aligned}
$$

we have the typing

$$\cdot \vdash (d\ k_1)(c\ k_1) : z$$

but not

$$\cdot \vdash (d\ k_1)(c\ k_2) : z$$

even though we have only changed one subterm of type a to another.

If we introduce proof irrelevance, however, then it can be shown that replacing one subterm under an irrelevant application with another preserves the whole term being well-typed. Therefore, it is safe to omit a subproof of a large proof if we can show that the subproof can be decidably recovered (i.e. if the predicate can be shown to be terminating) and occurs under an irrelevant application.

1.5 Extending Unification

The task before us is to modify the pattern unification algorithm to work in a language with proof irrelevance, and in particular to find the right notion of pattern. A similar situation arises with the notion of strict definitions [7], which depends on the definition of pattern spines. In the absence of irrelevance, a pattern spine is again a list of distinct bound variables. In [9] we found a modification of the definition of pattern spine which satisfies the same critical lemma as the original, again an injectivity property. The changes made were adding an 'irrelevant cons' to the syntax of spines and requiring distinct bound variables at 'relevant' positions and allowing any term at 'irrelevant' positions. We imitate these changes in a definition of pattern substitution, and observe that one-sided inverses again exist, for an appropriate notion of equality of substitutions. The theory here is simply-typed throughout, though we expect an extension to full dependent types to be reasonably straightforward.

2 Syntax

We define the $\lambda\sigma^{\iota}$-calculus, an extension of the typed $\lambda\sigma$-calculus with term variables [1] to include proof irrelevance in the sense of [6]. For uniformity of syntax, we have a sort of modalities μ with which we annotate function types, function abstractions, context and substitution conses, and applications (written $\diamond$). The modality $\mathbf{r}$ refers to the usual ('relevant') notion from λ-calculus and $\mathbf{i}$ gives the 'irrelevant' version.

Modalities	μ	$::= \mathbf{r} \mid \mathbf{i}$
Types	A, B	$::= \alpha \mid A \to^{\mu} B$
Contexts	Γ	$::= \bullet \mid A \cdot^{\mu} \Gamma$
Terms	a, b	$::= 1 \mid \mathcal{X} \mid (a \diamond^{\mu} b) \mid \lambda^{\mu}_{A} a \mid a[s]$
Substitutions	s, t	$::= \mathbf{id} \mid \uparrow \mid a \cdot^{\mu} s \mid s \circ t$

3 Typing

There are two typing judgments.

Term Typing	$\Gamma \vdash a :^{\mu} A$	a has type A for modality μ in the context Γ.
Substitution Typing	$\Gamma \vdash s : \Gamma'$	s is a substitution for the variables of Γ' in Γ.

Note that the term typing judgment is also annotated with a modality. We abbreviate $:^{\mathbf{r}}$ by $:$ and $:^{\mathbf{i}}$ by $\div$.

The meaning of the irrelevant typing judgment is given by the context operation $^{\oplus}$, defined recursively by

$$\bullet^{\oplus} = \bullet$$

$$(A \cdot^{\mu} \Gamma)^{\oplus} = A \cdot^{\mathbf{r}} (\Gamma^{\oplus})$$

and the rule

$$\frac{\Gamma^{\oplus} \vdash a : A}{\Gamma \vdash a \div A}$$

$\Gamma^{\oplus}$ is the context Γ with all irrelevant assumptions promoted to 'real' assumptions, so the typing rule allows us to conclude that $a \div A$ if we can derive $a : A$ when we are allowed to use even irrelevant assumptions.

The remaining typing rules are

$$\frac{}{A \cdot^{\mathbf{r}} \Gamma \vdash 1 : A} \qquad \frac{}{A \cdot^{\mu} \Gamma \vdash \uparrow : \Gamma}$$

$$\frac{A \cdot^{\mu} \Gamma \vdash b : B}{\Gamma \vdash \lambda^{\mu}_{A}.b : A \to^{\mu} B} \qquad \frac{\Gamma \vdash a : A \to^{\mu} B \qquad \Gamma \vdash b :^{\mu} B}{\Gamma \vdash (a \diamond^{\mu} b) : B}$$

$$\frac{}{\Gamma \vdash \mathbf{id} : \Gamma}$$

$$\frac{\Gamma \vdash s : \Gamma' \qquad \Gamma' \vdash a :^{\mu} A}{\Gamma \vdash a[s] :^{\mu} A}$$

$$\frac{\Gamma \vdash s : \Gamma' \qquad \Gamma' \vdash t : \Gamma''}{\Gamma \vdash t \circ s : \Gamma''}$$

$$\frac{\Gamma \vdash s : \Gamma' \qquad \Gamma \vdash a :^{\mu} A}{\Gamma \vdash a \cdot^{\mu} s : A \cdot^{\mu} \Gamma'}$$

$$\frac{}{\Gamma_{X} \vdash X : T_{X}}$$

4 Reduction

We list a set of reduction rules which are a straightforward adaptation of those in $\lambda\sigma$, including those added in [1] (**Id**, **IdR**, **VarShift**, **Scons**) for critical pairs arising from the addition of term variables.

Beta	$(\lambda^\mu_A.a)b$	$\to (a[b \cdot^\mu \mathbf{id}])$
Eta	$\lambda^\mu_A.(a \diamond^\mu 1)$	$\to b$ if $a =_\sigma b[\uparrow]$
App	$(a \diamond^\mu b)[s]$	$\to (a[s] \diamond^\mu b[s])$
VarCons	$1[a \cdot^\mu s]$	$\to a$
Id	$a[\mathbf{id}]$	$\to a$
Abs	$(\lambda^\mu_A.a)[s]$	$\to \lambda^\mu_A.(a[1 \cdot^\mu (s \circ \uparrow)])$
Clos	$(a[s])[t]$	$\to a[s \circ t]$
IdL	$\mathbf{id} \circ s$	$\to s$
IdR	$s \circ \mathbf{id}$	$\to s$
ShiftCons	$\uparrow \circ (a \cdot^\mu s)$	$\to s$
AssEnv	$(s_1 \circ s_2) \circ s_3$	$\to s_1 \circ (s_2 \circ s_3)$
MapEnv	$(a \cdot^\mu s) \circ t$	$\to a[t] \cdot^\mu (s \circ t)$
VarShift	$1 \cdot^\mu \uparrow$	$\to \mathbf{id}$
Scons	$1[s] \cdot^\mu (\uparrow \circ s)$	$\to s$

It remains to work out what normalization and confluence properties this system enjoys. It seems likely that weak or strong normalization of any fragment should follow easily from the same property holding of the corresponding fragment in the underlying calculus without irrelevance by a simple erasure of modality information. However, to whatever extent the property would depend on the typability of a term, we might encounter difficulty dealing with full dependent types, since erasing irrelevance from a dependently typed term may not result in a well-typed term.

We might also consider adding a reduction rule

$$\textbf{Irrel} \qquad a \cdot^{\mathbf{i}} s \to a' \cdot^{\mathbf{i}} s$$

to capture the intended meaning of an irrelevant cons, but this clearly destroys any hope of weak normalization. We expect to handle the necessary quotienting-out of terms at irrelevant conses in a different way.

5 Pattern substitutions

We define a judgment $s\,\mathbf{pat}^{n\geq I}$, where s is a substitution, n a natural number, and I a list of natural numbers. Its intended meaning is that s is a pattern substitution using deBruijn indices from I which are no greater than n.

$$\frac{}{\uparrow^n \mathbf{pat}^{n\geq\cdot}}$$

$$\frac{s\,\mathbf{pat}^{n\geq I} \qquad m \leq n, m \notin I}{\mathrm{m} \cdot^{\mathbf{r}} s\,\mathbf{pat}^{n\geq I,m}} \qquad \frac{s\,\mathbf{pat}^{n\geq I}}{a \cdot^{\mathbf{i}} s\,\mathbf{pat}^{n\geq I}}$$

From this we can define *pattern terms* via a judgment $a\,\mathbf{pat}^{n\geq I}$, defined by the rules

$$\frac{s\,\mathbf{pat}^{n\geq I}}{X[s]\,\mathbf{pat}^{n\geq I}}$$

$$\frac{a\,\mathbf{pat}^{n\geq I} \qquad m \leq n, m \notin I}{a \diamond^{\mathbf{r}} \mathrm{m}\,\mathbf{pat}^{n\geq I,m}} \qquad \frac{a\,\mathbf{pat}^{n\geq I}}{a \diamond^{\mathbf{i}} b\,\mathbf{pat}^{n\geq I}}$$

Definition 1. *A $\lambda\sigma'$-term is a* $\lambda\sigma'$-pattern *if all of its subterms of the form $a = (X[s]\ b_1 \cdots b_n)$ are such that a $\mathbf{pat}^{n \geq I}$ for some n, I. A* pattern substitution *is a $\lambda\sigma'$-substitution all of whose $\lambda\sigma'$-normal forms s are such that s $\mathbf{pat}^{n \geq I}$ for some n, I.*

Note that in the absence of any confluence result we hedge our definition of of pattern substitutions by referring to all normal forms rather than *the* normal form of the substitution s. Since we do not have substitution variables, we hope that in fact the normal forms of substitutions are well-behaved, and this inelegance can be removed.

The proof in [3] that pattern spines have one-sided inverses is constructive, and so can be described algorithmically. We first give a direct description, and then sketch how to modify the algorithm. By unrolling the induction definition, a pattern substitution ξ is of the form

$$a_1 \cdot^{\mu_1} a_2 \cdot^{\mu_2} \cdots a_n \cdot^{\mu_n} \uparrow^m$$

Put

$$\bar{\xi} = b_1 \cdot^{\mathbf{r}} b_2 \cdot^{\mathbf{r}} \cdots b_m \cdot^{\mathbf{r}} \uparrow^n$$

where

$$b_i = \begin{cases} \mathtt{j} & \text{If } \exists j. \mu_j = \mathbf{r} \wedge a_j = \mathtt{i}; \\ \text{a fresh variable } X & \text{otherwise.} \end{cases}$$

Observe that

$$\xi \circ \bar{\xi} \to^* a_1[\bar{\xi}] \cdot^{\mu_1} a_2[\bar{\xi}] \cdot^{\mu_2} \cdots a_n[\bar{\xi}] \cdot^{\mu_n} \uparrow^n$$

For any i, if μ_i is $\mathbf{r}$, then by definition of $\bar{\xi}$ and pattern substitutions we have $a_i[\bar{\xi}] \to^* \mathtt{i}$. Otherwise, μ_i is $\mathbf{i}$, and we have some term $a_i[\bar{\xi}]$ occurring at an irrelevant position in the substitution ξ. The intended meaning of irrelevance is that this is just as good as any other term at the same type, in particular the deBruijn index $\mathtt{i}$. Therefore, in a certain sense (and determining the right way to formalize this is the subject of our current effort) we have

$$\xi \circ \bar{\xi} \approx \mathtt{1} \cdot^{\mu_1} \mathtt{2} \cdot^{\mu_2} \cdots \mathtt{n} \cdot^{\mu_n} \uparrow^n \to^* \mathbf{id}$$

To modify the algorithm, we add modality annotations to the existing rules in a straightforward way, and add two rules,

$$\begin{array}{ll} \textbf{NDI} & n[a \cdot^{\mathbf{i}} \xi]^{-1} = (n[\xi]^{-1})[\uparrow] \\ \textbf{UDI} & \uparrow^m \circ (a \cdot^{\mathbf{i}} \xi)^{-1} = (\uparrow^m \circ \xi^{-1}) \circ \uparrow \end{array}$$

which cause irrelevant conses to act as if they were deBruijn indices that 'never occur,' in the sense that they never match other indices during application of an inverse substitution to a term (**NDI** is like **ND$\neq$** in [3]) and they never occur in the range of $\uparrow^m$ during composition of a substitution with an inverse substitution (**UDI** is like **UD1**).

6 Future Work

Computing the inverse is not the only operation on substitutions involved in the unification algorithm of [3]. We must also extend the definition of intersection $\xi \cap \zeta$ to transform equations of the form $X[\xi] = X[\zeta]$, and the pruning substitution $\xi|\zeta$ to correctly handle flexible occurrences of metavariables. Moreover it may be that the notion of flexible occurrence can be extended to include occurrences of metavariables anywhere under irrelevant application while maintaining the correctness of the overall algorithm. We intend to work out the appropriate extensions of these concepts towards a complete unification algorithm for higher-order patterns in the extended theory, and answer the basic questions of normalization and confluence mentioned above.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.
2. Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 366–374, San Diego, California, June 1995. IEEE Computer Society Press.
3. Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.
4. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
5. George C. Necula and Peter Lee. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.
6. Frank Pfenning. Intensionality, extensionality and proof irrelevance in modal type theory. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, 2001.
7. Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 179–193, Kloster Irsee, Germany, March 1998. Springer-Verlag LNCS 1657.
8. Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
9. Jason Reed. Proof irrelevance and strict definitions in a logical framework. Unpublished Manuscript, 2002.

A Verification of Rijndael in HOL

Konrad Slind

School of Computing, University of Utah

Abstract. We present a verification of the *Rijndael* symmetric block cipher in the HOL-4 theorem prover. In general, the proofs were easy: they proceeded largely by symbolic execution along with a few applications of algebraic rewrite rules, which were also easy to prove. However, the proofs depend on tight control of symbolic execution; otherwise, the problem size became too large for an interactive system. An important aspect of the formalization was to phrase Rijndael as a functional program.

1 Introduction

Rijndael [1, 2] is a collection of algorithms that encrypt and decrypt data. It recently won the AES (Advanced Encryption Standard) competition to find a successor to DES. It was designed to be suitable for implementation in software and hardware (from smartcards to full custom VLSI).

One of the attractions of verifying the functional correctness of such a system is the simplicity of its specification:

$$\forall key.\ \mathbf{decrypt}\ key \circ \mathbf{encrypt}\ key = \mathbf{I}$$

Of course, the essential further requirement of a cipher is that it be hard to break: decryption of encrypted data should be infeasible in the absence of the key used to encrypt. Our work does not address this problem, which appears far more difficult to settle. The usual methodology seems to be one of falsification: proposed ciphers are subjected to a variety of attacks; if none work, the cipher is deemed "secure", at least for the time being.

The two specification documents for Rijndael, one by its authors Rijmen and Daemen, and one from the AES organizers, are admirably done. Each step in the algorithm is explained carefully and there are useful glossaries aimed at avoiding any possible confusion. As well, there are appendices giving exact values for the intermediate values of the state after each of the important steps in a sample computation. However, for verification purposes, much of the specification has an unfortunate emphasis on arrays and updates on them. We therefore translated Rijndael into a purely functional program.

2 Technical Preliminaries

Many of the operations applied in Rijndael are operations involving the Galois field $\mathbf{GF}(2^8)$. This has a carrier set of 256 items, which is the number of elements enumerated by 8 bits. The byte $b_7b_6b_5b_4b_3b_2b_1b_0$ is considered as a polynomial with coefficients in $\{0, 1\}$:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

Polynomial addition is bitwise exclusive-or, and so is subtraction. Multiplication of polynomials is harder, since we need to be closed under the operation. Thus multiplication is performed modulo an irreducible polynomial of degree 8; the chosen one for Rijndael is

$$\mathbf{m}(x) = x^8 + x^4 + x^3 + x + 1$$

Written in hexadecimal, this is **11B**. Naively done, modular multiplication of polynomials is slow (multiplying through, then running a division algorithm) but it turns out that the full operation isn't needed: instead, multiplication by a constant suffices. In order to multiply a polynomial $b(x)$ by x, *i.e.*, by hex **02**, we can left shift ($\ll$) followed by a conditional **xor** with **11B** (which truncates to **1B**). This is the **xtime** operation:[1]

$$\textbf{xtime}\ b = (b \ll 1)\ \textbf{xor}\ (\texttt{if}\ b < 80\ \texttt{then}\ 0\ \texttt{else}\ 1B)$$

With this primitive we can multiply a polynomial by any constant: the operation will be written as an infix $\bullet$. Multiplication with higher powers of x can be achieved by iterating **xtime** and intermediate results can be added with **xor**: for example, multiplying **57** by **13** [2] yields

$$\begin{aligned} \mathbf{57} \bullet \mathbf{13} &= \mathbf{57} \bullet (\mathbf{01\ xor\ 02\ xor\ 10}) \\ &= \mathbf{57\ xor\ xtime(57)\ xor\ xtime(xtime(xtime(xtime(57))))} \\ &= \mathbf{57\ xor\ AE\ xor\ 07} \\ &= \mathbf{FE} \end{aligned}$$

There is also another notion of multiplication involved in Rijndael: one where the polynomials have *coefficients* in $\mathbf{GF}(2^8)$, *i.e.*, are bytes. However, that notion serves mainly at the specification level for the algorithm and does not manifest itself in the code. Similarly, since we are dealing with a field, there are also multiplicative inverse operations, but they are also not explicit in the code of the algorithm and so will not be discussed.

3 Rijndael as a functional program

We will present Rijndael as a program in SML. Translation to other functional programming languages should be easy. Since the algorithm deals extensively with bits and bytes, it is helpful if the host programming language supports operations on these types. The SML library provides a structure **Word8** implementing bytes (the type **word8**). Literals for bytes may be written in hexadecimal in the format **0wx**h_1h_2. Exclusive-or is provided by **xorb**. Rijndael is defined for three keylengths: 128, 192, and 256 bits. Our verification is for a keylength of 128.

3.1 The state

The algorithm operates by repeatedly transforming a *state* of 16 8-bit bytes. In the original specifications, the state is represented as a 4x4 array. The algorithms access the state by byte, by row, and by column. Instead of an array, we represent a state by a 16-tuple of bytes.

```
type state = word8 * word8 * word8 * word8 *
             word8 * word8 * word8 * word8 *
             word8 * word8 * word8 * word8 *
             word8 * word8 * word8 * word8
```

The plaintext input $b_0b_1b_2b_3b_4b_5b_6b_7b_8b_9b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}$ is moved into the state by proceeding from left to right through the input and placing the bytes into 'columns'. The inverse operation is used in decryption.

```
fun to_state (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
             =
             (b0,b4,b8,b12,
              b1,b5,b9,b13,
              b2,b6,b10,b14,
              b3,b7,b11,b15)

fun from_state (b0,b4,b8,b12,
                b1,b5,b9,b13,
                b2,b6,b10,b14,
                b3,b7,b11,b15) = (b0,b1,b2,b3,b4,b5,b6,b7,b8,
                                  b9,b10,b11,b12,b13,b14,b15)
```

These functions show how pattern matching on tuples is used throughout instead of array indexing.

[1] Note that the $<$ in this definition is a comparison on bytes, and the literals 80, 0, and 1B are hexadecimal.

[2] All literals are hexadecimal

3.2 Rounds

The main steps of the algorithm are orchestrated by the *round* computation. We have phrased this and its inverse as recursive functions.

```
fun Round 0 [key] state = AddRoundKey key (ShiftRows (SubBytes state))
  | Round n (key::keys) state =
     Round (n-1) keys
        (AddRoundKey key
            (MixColumns (ShiftRows (SubBytes state))))
  | Round _ _ _ = raise Fail "Round: bug";

fun InvRound 0 [key] state = AddRoundKey key
                                (InvSubBytes(InvShiftRows state))
  | InvRound n (key::keys) state =
     InvRound (n-1) keys
        (InvMixColumns
           (AddRoundKey key
             (InvSubBytes (InvShiftRows state))))
  | InvRound _ _ _ = raise Fail "InvRound: bug";
```

In a round, the main operations on the state are to perform byte substitution using so called *sboxes*, to shift the rows of the state, and to mix the columns of the state. We will discuss these in turn.

Sboxes The *Sbox* is a permutation on bytes designed to be resistant to linear and differential cryptanalysis. We create the function and its inverse **InvSbox** from vectors of bytes.

```
val Sbox = curry Vector.sub (Vector.fromList
  [0wx63,0wx7c,0wx77,0wx7b,0wxf2,0wx6b,0wx6f,0wxc5,0wx30,0wx01,0wx67,0wx2b,0wxfe,0wxd7,0wxab,0wx76,
   0wxca,0wx82,0wxc9,0wx7d,0wxfa,0wx59,0wx47,0wxf0,0wxad,0wxd4,0wxa2,0wxaf,0wx9c,0wxa4,0wx72,0wxc0,
   0wxb7,0wxfd,0wx93,0wx26,0wx36,0wx3f,0wxf7,0wxcc,0wx34,0wxa5,0wxe5,0wxf1,0wx71,0wxd8,0wx31,0wx15,
   0wx04,0wxc7,0wx23,0wxc3,0wx18,0wx96,0wx05,0wx9a,0wx07,0wx12,0wx80,0wxe2,0wxeb,0wx27,0wxb2,0wx75,
   0wx09,0wx83,0wx2c,0wx1a,0wx1b,0wx6e,0wx5a,0wxa0,0wx52,0wx3b,0wxd6,0wxb3,0wx29,0wxe3,0wx2f,0wx84,
   0wx53,0wxd1,0wx00,0wxed,0wx20,0wxfc,0wxb1,0wx5b,0wx6a,0wxcb,0wxbe,0wx39,0wx4a,0wx4c,0wx58,0wxcf,
   0wxd0,0wxef,0wxaa,0wxfb,0wx43,0wx4d,0wx33,0wx85,0wx45,0wxf9,0wx02,0wx7f,0wx50,0wx3c,0wx9f,0wxa8,
   0wx51,0wxa3,0wx40,0wx8f,0wx92,0wx9d,0wx38,0wxf5,0wxbc,0wxb6,0wxda,0wx21,0wx10,0wxff,0wxf3,0wxd2,
   0wxcd,0wx0c,0wx13,0wxec,0wx5f,0wx97,0wx44,0wx17,0wxc4,0wxa7,0wx7e,0wx3d,0wx64,0wx5d,0wx19,0wx73,
   0wx60,0wx81,0wx4f,0wxdc,0wx22,0wx2a,0wx90,0wx88,0wx46,0wxee,0wxb8,0wx14,0wxde,0wx5e,0wx0b,0wxdb,
   0wxe0,0wx32,0wx3a,0wx0a,0wx49,0wx06,0wx24,0wx5c,0wxc2,0wxd3,0wxac,0wx62,0wx91,0wx95,0wxe4,0wx79,
   0wxe7,0wxc8,0wx37,0wx6d,0wx8d,0wxd5,0wx4e,0wxa9,0wx6c,0wx56,0wxf4,0wxea,0wx65,0wx7a,0wxae,0wx08,
   0wxba,0wx78,0wx25,0wx2e,0wx1c,0wxa6,0wxb4,0wxc6,0wxe8,0wxdd,0wx74,0wx1f,0wx4b,0wxbd,0wx8b,0wx8a,
   0wx70,0wx3e,0wxb5,0wx66,0wx48,0wx03,0wxf6,0wx0e,0wx61,0wx35,0wx57,0wxb9,0wx86,0wxc1,0wx1d,0wx9e,
   0wxe1,0wxf8,0wx98,0wx11,0wx69,0wxd9,0wx8e,0wx94,0wx9b,0wx1e,0wx87,0wxe9,0wxce,0wx55,0wx28,0wxdf,
   0wx8c,0wxa1,0wx89,0wx0d,0wxbf,0wxe6,0wx42,0wx68,0wx41,0wx99,0wx2d,0wx0f,0wxb0,0wx54,0wxbb,0wx16])
  o Word8.toInt

val InvSbox = curry Vector.sub (Vector.fromList
  [0wx52,0wx09,0wx6a,0wxd5,0wx30,0wx36,0wxa5,0wx38,0wxbf,0wx40,0wxa3,0wx9e,0wx81,0wxf3,0wxd7,0wxfb,
   0wx7c,0wxe3,0wx39,0wx82,0wx9b,0wx2f,0wxff,0wx87,0wx34,0wx8e,0wx43,0wx44,0wxc4,0wxde,0wxe9,0wxcb,
   0wx54,0wx7b,0wx94,0wx32,0wxa6,0wxc2,0wx23,0wx3d,0wxee,0wx4c,0wx95,0wx0b,0wx42,0wxfa,0wxc3,0wx4e,
   0wx08,0wx2e,0wxa1,0wx66,0wx28,0wxd9,0wx24,0wxb2,0wx76,0wx5b,0wxa2,0wx49,0wx6d,0wx8b,0wxd1,0wx25,
   0wx72,0wxf8,0wxf6,0wx64,0wx86,0wx68,0wx98,0wx16,0wxd4,0wxa4,0wx5c,0wxcc,0wx5d,0wx65,0wxb6,0wx92,
   0wx6c,0wx70,0wx48,0wx50,0wxfd,0wxed,0wxb9,0wxda,0wx5e,0wx15,0wx46,0wx57,0wxa7,0wx8d,0wx9d,0wx84,
   0wx90,0wxd8,0wxab,0wx00,0wx8c,0wxbc,0wxd3,0wx0a,0wxf7,0wxe4,0wx58,0wx05,0wxb8,0wxb3,0wx45,0wx06,
   0wxd0,0wx2c,0wx1e,0wx8f,0wxca,0wx3f,0wx0f,0wx02,0wxc1,0wxaf,0wxbd,0wx03,0wx01,0wx13,0wx8a,0wx6b,
   0wx3a,0wx91,0wx11,0wx41,0wx4f,0wx67,0wxdc,0wxea,0wx97,0wxf2,0wxcf,0wxce,0wxf0,0wxb4,0wxe6,0wx73,
   0wx96,0wxac,0wx74,0wx22,0wxe7,0wxad,0wx35,0wx85,0wxe2,0wxf9,0wx37,0wxe8,0wx1c,0wx75,0wxdf,0wx6e,
   0wx47,0wxf1,0wx1a,0wx71,0wx1d,0wx29,0wxc5,0wx89,0wx6f,0wxb7,0wx62,0wx0e,0wxaa,0wx18,0wxbe,0wx1b,
   0wxfc,0wx56,0wx3e,0wx4b,0wxc6,0wxd2,0wx79,0wx20,0wx9a,0wxdb,0wxc0,0wxfe,0wx78,0wxcd,0wx5a,0wxf4,
   0wx1f,0wxdd,0wxa8,0wx33,0wx88,0wx07,0wxc7,0wx31,0wxb1,0wx12,0wx10,0wx59,0wx27,0wx80,0wxec,0wx5f,
   0wx60,0wx51,0wx7f,0wxa9,0wx19,0wxb5,0wx4a,0wx0d,0wx2d,0wxe5,0wx7a,0wx9f,0wx93,0wxc9,0wx9c,0wxef,
   0wxa0,0wxe0,0wx3b,0wx4d,0wxae,0wx2a,0wxf5,0wxb0,0wxc8,0wxeb,0wxbb,0wx3c,0wx83,0wx53,0wx99,0wx61,
   0wx17,0wx2b,0wx04,0wx7e,0wxba,0wx77,0wxd6,0wx26,0wxe1,0wx69,0wx14,0wx63,0wx55,0wx21,0wx0c,0wx7d])
  o Word8.toInt
```

3.3 Byte Substitution

A byte substitution step applies an sbox to each element in the state: we phrase this as a higher-order function for re-use:

```
fun genSubBytes S (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                              =
      (S b0, S b1, S b2, S b3, S b4, S b5, S b6, S b7,
       S b8, S b9, S b10, S b11, S b12, S b13, S b14, S b15)

val SubBytes    = genSubBytes Sbox
val InvSubBytes = genSubBytes InvSbox
```

3.4 Shift Rows

In a row shift step, the first row is not altered, the second row is left-shifted by one, the third row is left-shifted by 2, and the fourth row is left-shifted by 3.

```
fun ShiftRows (b00,b01,b02,b03,
               b10,b11,b12,b13,
               b20,b21,b22,b23,
               b30,b31,b32,b33)
                     =
              (b00,b01,b02,b03,
               b11,b12,b13,b10,
               b22,b23,b20,b21,
               b33,b30,b31,b32)

fun InvShiftRows (b00,b01,b02,b03,
                  b11,b12,b13,b10,
                  b22,b23,b20,b21,
                  b33,b30,b31,b32)
                        =
                 (b00,b01,b02,b03,
                  b10,b11,b12,b13,
                  b20,b21,b22,b23,
                  b30,b31,b32,b33)
```

3.5 Mix Columns

The mixing of columns is relatively complex in its operation. Each column in the state is treated as a four-term polynomial over **GF**(2^8) and multiplied modulo x^4+1 with a fixed polynomial. A higher-order function captures the general pattern for the forward and reverse operations:

```
fun genMixColumns MC (b00,b01,b02,b03,
                      b10,b11,b12,b13,
                      b20,b21,b22,b23,
                      b30,b31,b32,b33)
  = let val (b00', b10', b20', b30') = MC (b00,b10,b20,b30)
        val (b01', b11', b21', b31') = MC (b01,b11,b21,b31)
        val (b02', b12', b22', b32') = MC (b02,b12,b22,b32)
        val (b03', b13', b23', b33') = MC (b03,b13,b23,b33)
    in
     (b00', b01', b02', b03',
      b10', b11', b12', b13',
      b20', b21', b22', b23',
      b30', b31', b32', b33')
    end

val MixColumns = genMixColumns MultCol
val InvMixColumns = genMixColumns InvMultCol
```

In the forward direction, the fixed polynomial is $a(x) = \mathbf{03}x^3 + \mathbf{01}x^2 + \mathbf{01}x + \mathbf{02}$. After some massaging, we arrive at the following transformation on the column:

```
fun MultCol (a,b,c,d) =
 ((0wx02 ** a) xorb (0wx03 ** b) xorb  c          xorb d,
   a           xorb (0wx02 ** b) xorb (0wx03 ** c) xorb d,
   a           xorb  b           xorb (0wx02 ** c) xorb (0wx03 ** d),
  (0wx03 ** a) xorb  b           xorb  c           xorb (0wx02 ** d))
```

The inverse operation is harder, in the sense that larger coefficients are used: the fixed polynomial is $a^{-1}(x) = \mathbf{0B}x^3 + \mathbf{0D}x^2 + \mathbf{09}x + \mathbf{0E}$. The column transformation is:

```
fun InvMultCol (a,b,c,d) =
 ((0wx0e ** a) xorb (0wx0b ** b) xorb (0wx0d ** c) xorb (0wx09 ** d),
  (0wx09 ** a) xorb (0wx0e ** b) xorb (0wx0b ** c) xorb (0wx0d ** d),
  (0wx0d ** a) xorb (0wx09 ** b) xorb (0wx0e ** c) xorb (0wx0b ** d),
  (0wx0b ** a) xorb (0wx0d ** b) xorb (0wx09 ** c) xorb (0wx0e ** d))
```

These operations are defined in terms of multiplication by a constant, represented by the infix ** symbol:

```
fun xtime b = (b << 0wx1) xorb (if b < 0wx80 then 0wx0 else 0wx1B)

fun (0wx0 ** v) = 0wx0
  | (c    ** v) = if andb(c,0wx01) = 0wx01
                    then v xorb ((c >> 0wx1) ** (xtime v))
                    else        ((c >> 0wx1) ** (xtime v))
```

There is also an exponentiation operation, used to generate the key schedule:

```
fun exp (x,0) = 0wx01
  | exp (x,n) = x ** exp (x,n-1)
```

4 Generating the key schedule

An important part of Rijndael is the calculation of the *key schedule* (a list of *round keys*) from the original key, as a preliminary step to the round computations. In each round, a new round key is added to the state pointwise with AddRoundKey:

```
fun AddRoundKey
   (k0,k1,k2,k3,k4,k5,k6,k7,k8,k9,k10,k11,k12,k13,k14,k15)
   (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                         =
   (b0 xorb k0, b1 xorb k1, b2 xorb k2, b3 xorb k3,
    b4 xorb k4, b5 xorb k5, b6 xorb k6, b3 xorb k3,
    b8 xorb k8, b9 xorb k9, b10 xorb k10, b11 xorb k11,
    b12 xorb k12, b13 xorb k13, b14 xorb k14, b15 xorb k15)
```

The specification calls for the key schedule to be generated by operations on 32-bit words. In our version of ML (Moscow ML), only 31-bit words were available, so we rephrased the algorithm over quadruples of bytes. We will pass over the code in silence, since it is somewhat involved and the details are not important for the correctness proof.

```
local open Int   nonfix o
      fun SubWord(b0,b1,b2,b3) = (Sbox b0, Sbox b1, Sbox b2, Sbox b3)
      fun RotWord(b0,b1,b2,b3) = (b1,b2,b3,b0)
      fun Rcon i = (exp(0wx02, i-1), 0wx00,0wx00,0wx00) : word8x4
      fun unpack [] A = A
        | unpack ((a,b,c,d)::(e,f,g,h)::(i,j,k,l)::(m,n,o,p)::rst) A
```

```
          = unpack rst ((m,i,e,a,n,j,f,b,o,k,g,c,p,l,h,d)::A)
       | unpack otherwise _ = raise Fail "generate_keysched(unpack)"
in
fun mk_keysched top (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15) =
 let fun expand n (sched as (h::_::_::last::_)) =
      if n>top then unpack sched []
       else let val h' = if n mod 4 <> 0 then h
                         else SubWord(RotWord h) xor4 Rcon(n div 4)
             in expand (n+1) ((h' xor4 last)::sched)
             end
  in
     expand 4 [(b12,b13,b14,b15),(b8,b9,b10,b11),(b4,b5,b6,b7),(b0,b1,b2,b3)]
  end
end
```

Finally, the top-level functionality can be obtained by a function that takes a key and generates the key schedule before building the encryption and decryption functions. The encryption function uses the key schedule and the decryption function uses the inverse of the key schedule.

```
fun preCrypt key =
 let open Int
     val Nr = 10
     val keysched = mk_keysched (BlockSize * (Nr+1) -1) key
     val (key0::keys) = keysched
     val (ikey0::ikeys) = List.rev keysched
 in
    (from_state o Round (Nr-1) keys o AddRoundKey key0 o to_state,
     from_state o InvRound (Nr-1) ikeys o AddRoundKey ikey0 o to_state)
 end
```

5 The verification of Rijndael

Rijndael is directly encoded in HOL with only a few alterations from the SML program.

5.1 Bytes

An interesting modelling question is how best to represent bytes. In SML, bytes (`word8`) are an abstract type, enumerated by 256 literals. HOL does not however have bytes built in, so we were confronted with several choices: bytes may be represented by an enumerated type, or by the numbers up to 256, or by 8-tuples of truth values. Although it is inefficient in a sense, we chose the latter representation.

To start, we define a few byte constants:

$$\begin{aligned}
\textbf{ZERO} &= (F,F,F,F,F,F,F,F)\\
\textbf{ONE} &= (F,F,F,F,F,F,F,T)\\
\textbf{TWO} &= (F,F,F,F,F,F,T,F)\\
\textbf{THREE} &= (F,F,F,F,F,F,T,T)\\
\textbf{NINE} &= (F,F,F,F,T,F,F,T)\\
\textbf{ONE\_B} &= (F,F,F,T,T,F,T,T)\\
\textbf{EIGHTY} &= (T,F,F,F,F,F,F,F)\\
\textbf{B} &= (F,F,F,F,T,F,T,T)\\
\textbf{D} &= (F,F,F,F,T,T,F,T)\\
\textbf{E} &= (F,F,F,F,T,T,T,F)
\end{aligned}$$

Infix operators for 'exclusive-or' on bits and bytes are defined, along with an infix 'and' operation on bytes:

$$x \textbf{ xor } y = \neg(x = y)$$

$$\begin{pmatrix} (a, b, c, d, e, f, g, h) \\ \textbf{XOR} \\ (a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1) \end{pmatrix} = \begin{pmatrix} a \textbf{ xor } a_1, b \textbf{ xor } b_1, c \textbf{ xor } c_1, d \textbf{ xor } d_1, \\ e \textbf{ xor } e_1, f \textbf{ xor } f_1, g \textbf{ xor } g_1, h \textbf{ xor } h_1 \end{pmatrix}$$

$$\begin{pmatrix} (a, b, c, d, e, f, g, h) \\ \textbf{AND} \\ (a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1) \end{pmatrix} = \begin{pmatrix} a \land a_1, b \land b_1, c \land c_1, d \land d_1, \\ e \land e_1, f \land f_1, g \land g_1, h \land h_1 \end{pmatrix}$$

A few trivial algebraic theorems then follow: **ZERO** is the identity for **XOR**, commutativity, and associativity.

$$\vdash x \textbf{ XOR ZERO} = x$$
$$\vdash (x \textbf{ XOR } y) = (y \textbf{ XOR } x)$$
$$\vdash (x \textbf{ XOR } y) \textbf{ XOR } z = x \textbf{ XOR } (y \textbf{ XOR } z)$$

5.2 The state

The functions **to_state** and **from_state** for mapping into and out of a state are exactly the same as the ML definitions. That they are inverses of each other is trivial:

$$\vdash \forall s.\ \textbf{from\_state}(\textbf{to\_state}\ s) = s$$
$$\vdash \forall s.\ \textbf{to\_state}(\textbf{from\_state}\ s) = s$$

5.3 Applying an sbox to the state

The functions **SubBytes** and **InvSubBytes** for applying an sbox to a state are exactly the same as the ML definitions. The sboxes are each defined by a 256-way pattern match. The inversion theorem for these functions is a consequence of the inversion theorem for sboxes, which is proved by analyzing all 256 cases and evaluating the sboxes.

$$\vdash \forall w.\ \textbf{InvSbox}(\textbf{Sbox}\ w) = w$$
$$\vdash \forall s.\ \textbf{InvSubBytes}(\textbf{SubBytes}\ s) = s$$

5.4 Shifting rows

The functions **ShiftRows** and **InvShiftRows** for shifting rows in a state are exactly the same as the ML definitions. The inversion theorem for these functions is trivial to prove.

$$\vdash \forall s.\ \textbf{InvShiftRows}(\textbf{ShiftRows}\ s) = s$$

5.5 Multiplication

The definitions of the multiplication functions largely follow the definitions. The **xtime** function is slightly different, owing to our representation:

```
$\textbf{xtime}\,(b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$ =
    if $b_7$ then $(b_6, b_5, b_4, \neg b_3, \neg b_2, b_1, \neg b_0, T)$
          else $(b_6, b_5, b_4, b_3, b_2, b_1, b_0, F)$
```

The **xtime** function enjoys a distributive property:

$$\forall a\ b.\ \textbf{xtime}(a \textbf{ XOR } b) = (\textbf{xtime}\ a) \textbf{ XOR } (\textbf{xtime}\ b)$$

Multiplication by a constant is a direct translation.

```
b₁ • b₂ = if b₁ = ZERO then ZERO else
          if (b₁ AND ONE) = ONE
             then b₂ XOR ((RightShift b₁) • (xtime b₂))
             else        (RightShift b₁) • (xtime b₂)
```

Termination of the function is proved by regarding the first argument as a number. The • operation distributes over **XOR**:

$$\vdash \forall x\ y\ z.\ x \bullet (y \ \mathbf{XOR}\ z) = (x \bullet y)\ \mathbf{XOR}\ (x \bullet z)$$

5.6 Column mixing

With column mixing, the proofs became larger. We need the inversion theorem

$$\vdash \forall s : \mathit{state}.\ \mathbf{InvMixColumns}(\mathbf{MixColumns}\ s) = s$$

for the final proof, but naive case analyses became too large and we had to resort to much more basic steps. To see the problem let's consider the action on a column (a, b, c, d). In the forward direction, we have

$$\begin{aligned} a' &= F_1(a, b, c, d) \\ b' &= F_2(a, b, c, d) \\ c' &= F_3(a, b, c, d) \\ d' &= F_4(a, b, c, d) \end{aligned}$$

and in the reverse we build up

$$\begin{aligned} a'' &= G_1(a', b', c', d') \\ b'' &= G_2(a', b', c', d') \\ c'' &= G_3(a', b', c', d') \\ d'' &= G_4(a', b', c', d') \end{aligned}$$

and we wish to show that $a = a'', b = b'', c = c'', d = d''$. Consideration of a should illustrate our strategy.

$$\begin{aligned} a' &= (\mathbf{TWO} \bullet a)\ \mathbf{XOR}\ (\mathbf{THREE} \bullet b)\ \mathbf{XOR}\ c\ \mathbf{XOR}\ d \\ b' &= a\ \mathbf{XOR}\ (\mathbf{TWO} \bullet b)\ \mathbf{XOR}\ (\mathbf{THREE} \bullet c)\ \mathbf{XOR}\ d \\ c' &= a\ \mathbf{XOR}\ b\ \mathbf{XOR}\ (\mathbf{TWO} \bullet c)\ \mathbf{XOR}\ (\mathbf{THREE} \bullet d) \\ d' &= (\mathbf{THREE} \bullet a)\ \mathbf{XOR}\ b\ \mathbf{XOR}\ c\ \mathbf{XOR}\ (\mathbf{TWO} \bullet d) \end{aligned}$$

Thus

$$\begin{aligned} a'' &= (\mathbf{E} \bullet a')\ \mathbf{XOR}\ (\mathbf{B} \bullet b')\ \mathbf{XOR}\ (\mathbf{D} \bullet c')\ \mathbf{XOR}\ (\mathbf{NINE} \bullet d') \\ &= (\mathbf{E} \bullet ((\mathbf{TWO} \bullet a)\ \mathbf{XOR}\ (\mathbf{THREE} \bullet b)\ \mathbf{XOR}\ c\ \mathbf{XOR}\ d))\ \mathbf{XOR} \\ &\quad\ (\mathbf{B} \bullet (a\ \mathbf{XOR}\ (\mathbf{TWO} \bullet b)\ \mathbf{XOR}\ (\mathbf{THREE} \bullet c)\ \mathbf{XOR}\ d))\ \mathbf{XOR} \\ &\quad\ (\mathbf{D} \bullet (a\ \mathbf{XOR}\ b\ \mathbf{XOR}\ (\mathbf{TWO} \bullet c)\ \mathbf{XOR}\ (\mathbf{THREE} \bullet d)))\ \mathbf{XOR} \\ &\quad\ (\mathbf{NINE} \bullet ((\mathbf{THREE} \bullet a)\ \mathbf{XOR}\ b\ \mathbf{XOR}\ c\ \mathbf{XOR}\ (\mathbf{TWO} \bullet d))) \end{aligned}$$

By use of associativity and commutativity of **XOR** and distribution of • over **XOR**, we can separate the problems into subproblems involving only one variable, each of which are easy to solve by case analysis on the 256 ways of forming a byte.

$$\begin{aligned} a'' = &\ (\mathbf{E} \bullet (\mathbf{TWO} \bullet a)\ \mathbf{XOR}\ \mathbf{E} \bullet a\ \mathbf{XOR}\ \mathbf{D} \bullet a\ \mathbf{XOR}\ \mathbf{NINE} \bullet (\mathbf{THREE} \bullet a)) \\ &\ \mathbf{XOR} \\ &\ (\mathbf{E} \bullet (\mathbf{THREE} \bullet b)\ \mathbf{XOR}\ \mathbf{E} \bullet (\mathbf{TWO} \bullet b)\ \mathbf{XOR}\ \mathbf{D} \bullet b\ \mathbf{XOR}\ \mathbf{NINE} \bullet b) \\ &\ \mathbf{XOR} \\ &\ (\mathbf{E} \bullet c\ \mathbf{XOR}\ \mathbf{E} \bullet (\mathbf{THREE} \bullet c)\ \mathbf{XOR}\ \mathbf{D} \bullet (\mathbf{TWO} \bullet c)\ \mathbf{XOR}\ \mathbf{NINE} \bullet c) \\ &\ \mathbf{XOR} \\ &\ (\mathbf{E} \bullet d\ \mathbf{XOR}\ \mathbf{E} \bullet d\ \mathbf{XOR}\ \mathbf{D} \bullet (\mathbf{THREE} \bullet d)\ \mathbf{XOR}\ \mathbf{NINE} \bullet (\mathbf{TWO} \bullet d)) \\ = &\ a\ \mathbf{XOR}\ \mathbf{ZERO}\ \mathbf{XOR}\ \mathbf{ZERO}\ \mathbf{XOR}\ \mathbf{ZERO} \\ = &\ a \end{aligned}$$

5.7 Joe Hurd's good suggestion

The computation of the key schedule is fairly complex. If we needed non-trivial properties of it in order to show correctness, significant extra work would be required. After listening to a preliminary presentation on this work, Joe Hurd suggested that it might suffice merely to treat the key schedule as an arbitrary list of keys. In fact, since the round computation consumes one key per round, it suffices to show that the key schedule is an arbitrary key list of length 11. With this fact, the final correctness proof had no more impediments.

To prove this was, again, not simply a matter of symbolic evaluation. First we proved an invariant on the key expansion routine.

$$\begin{array}{l} \vdash \forall n\ t. \\ \quad 3 < n \wedge n < 44 \\ \quad\quad \Rightarrow \\ \quad \exists h.\ \mathbf{expand}\ (n+1)\ (h :: t) = \mathbf{expand}\ n\ t \end{array}$$

This leads directly to a theorem relating the first and last calls of the key expansion routine.

$$\begin{array}{l} \vdash \forall a\ b\ c\ d. \\ \quad \exists h_1\ h_2\ h_3\ h_4\ h_5\ h_6\ h_7\ h_8\ h_9\ h_{10}\ h_{11}\ h_{12}\ h_{13}\ h_{14}\ h_{15}\ h_{16}\ h_{17}\ h_{18}\ h_{19}\ h_{20} \\ \quad\ h_{21}\ h_{22}\ h_{23}\ h_{24}\ h_{25}\ h_{26}\ h_{27}\ h_{28}\ h_{29}\ h_{30}\ h_{31}\ h_{32}\ h_{33}\ h_{34}\ h_{35}\ h_{36}\ h_{37}\ h_{38}\ h_{39}\ h_{40}. \\ \quad \mathbf{expand}\ 44\ [h_{40}; h_{39}; h_{38}; h_{37}; h_{36}; h_{35}; h_{34}; h_{33}; h_{32}; h_{31}; h_{30}; h_{29}; h_{28}; \\ \quad\quad\quad h_{27}; h_{26}; h_{25}; h_{24}; h_{23}; h_{22}; h_{21}; h_{20}; h_{19}; h_{18}; h_{17}; h_{16}; h_{15}; \\ \quad\quad\quad h_{14}; h_{13}; h_{12}; h_{11}; h_{10}; h_9; h_8; h_7; h_6; h_5; h_4; h_3; h_2; h_1; a; b; c; d] \\ \quad\quad = \\ \quad \mathbf{expand}\ 4\ [a; b; c; d] \end{array}$$

From this we quickly get that the length of the list returned by **mk_keysched** is 11.

$$\begin{array}{l} \vdash \forall key.\ \exists h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8 h_9 h_{10} h_{11}. \\ \quad \mathbf{mk\_keysched}\ key = [h_1; h_2; h_3; h_4; h_5; h_6; h_7; h_8; h_9; h_{10}; h_{11}] \end{array}$$

This theorem can be immediately used to build a representation of the key schedule suitable for symbolic execution.

5.8 Correctness

The statement of correctness is

```
∀key plaintext.
  let (encrypt, decrypt) = preCrypt key
  in
    decrypt(encrypt plaintext) = plaintext
```

The definition of **preCrypt** is merely an organizational device aimed at making a neat statement of the final theorem. In the definition, the key schedule and its reverse are built from the key, and then the pair of functions (*encrypt*, *decrypt*) is returned. The encryption function copies the input into the state, makes an initial scrambling step with the first key, and then makes 10 rounds of further scrambling before transferring the final state to the output. The proof shows that the decryption function basically reverses these steps.

```
preCrypt key =
   let sched = mk_keysched key in
   let isched = REVERSE sched
   in
    ((from_state ∘ Round 9 (TL sched)
                 ∘ AddRoundKey (HD sched) ∘ to_state),
     (from_state ∘ InvRound 9 (TL isched)
                 ∘ AddRoundKey (HD isched) ∘ to_state))
```

The proof of correctness starts by expanding the definition of **preCrypt**. Then the lemma on the length of key schedules is used to replace the variables representing the key schedule and its reverse by corresponding lists of eleven variables. Now the 10 rounds of encryption and the 10 rounds of decryption are unwound, giving a large formula. The proof finishes by rewriting with the inversion lemmas.

5.9 An alternative decryptor

Daemen and Rijmen present an alternative implementation of the inverse round computation:

```
fun EqInvRound 0 [key] state = AddRoundKey key
                                  (InvShiftRows
                                     (InvSubBytes state))
  | EqInvRound n (key::keys) state =
        EqInvRound (n-1) keys
           (AddRoundKey key
              (InvMixColumns
                 (InvShiftRows
                    (InvSubBytes state))))
```

The alternative differs from the original in that the calls to **InvShiftRow** and **InvSubBytes** are swapped, as are the calls to **AddRoundKey** and **InvMixColumns**.

InvRound with a key schedule *ks* is equivalent to **EqInvRound** with **InvMixColumn** mapped over *ks* (except for the first and last elements). The mapping operation over the key schedule is called **InvMixify**:

$$\begin{aligned}\textbf{InvMix}\ [x] &= [x]\\ \textbf{InvMix}\ (h :: t) &= \textbf{InvMixColumns}\ h :: \textbf{InvMix}\ t\end{aligned}$$

$$\textbf{InvMixify}\ (h :: t) = h :: \textbf{InvMix}\ t$$

In the alternative version of **preCrypt**, the forward computation is unchanged, and only the inverse rounds and their key schedule are altered:

$$\begin{array}{l}\textbf{preCryptAlt}\ \mathit{key} =\\ \quad \texttt{let}\ \mathit{sched} = \textbf{mk\_keysched}\ \mathit{key}\ \texttt{in}\\ \quad \texttt{let}\ \mathit{isched} = \textbf{InvMixify}\ (\textbf{REVERSE}\ \mathit{sched})\\ \quad \texttt{in}\\ \quad\ ((\textbf{from\_state} \circ \textbf{Round}\ 9\ (\textbf{TL}\ \mathit{sched})\\ \qquad\qquad \circ\ \textbf{AddRoundKey}\ (\textbf{HD}\ \mathit{sched}) \circ \textbf{to\_state}),\\ \quad\ \ (\textbf{from\_state} \circ \textbf{EqInvRound}\ 9\ (\textbf{TL}\ \mathit{isched})\\ \qquad\qquad \circ\ \textbf{AddRoundKey}\ (\textbf{HD}\ \mathit{isched}) \circ \textbf{to\_state}))\end{array}$$

With the lemmas

$$\begin{array}{l}\vdash \forall s.\ \textbf{InvShiftRows}\ (\textbf{InvSubBytes}\ s) = \textbf{InvSubBytes}\ (\textbf{InvShiftRows}\ s)\\ \vdash \forall s\ k.\ \textbf{InvMixColumns}\ (\textbf{AddRoundKey}\ s\ k)\\ \qquad\quad =\\ \qquad \textbf{AddRoundKey}\ (\textbf{InvMixColumns}\ s)\ (\textbf{InvMixColumns}\ k)\end{array}$$

it is easy to prove

$$\vdash \textbf{preCryptAlt} = \textbf{preCrypt}.$$

6 Conclusions

The verification of Rijndael was relatively easy, which is good. One aspect of the problem was learning — by trial and error — which definitions led to exponential symbolic evaluations. Lemmas about the generation of the key schedule and the mixing of columns were the main examples and required the majority of the effort. Symbolic execution allowed the avoidance of any intimidating abstract algebra. Rijndael, when rendered as a functional program, is also quite simple, and could be taught to undergraduates with little difficulty. Thus, it may be useful as a pedagogical example of verification technology.

There are several interesting avenues to explore:

- We anticipate that proofs for the other key lengths will be straightforward.
- The code we have proved correct encrypts and decrypts only a single block. So-called *modes of operation* specify various ways to encrypt arbitrary streams of data. Extending our work to these should be straightforward.
- We would like to investigate the generation of hardware, *e.g.*, gate arrays, directly from the HOL formulation. There has already been much work on putting Rijndael into hardware, but the provision of a path from higher-order logic to hardware seems appealing.
- Finally, encryption is one of a family of similar operations characterized by *invertibility*; for example, compression/decompression and encoding/decoding. It would be interesting to see if commonalities can be found in the correctness proofs of these algorithms.

References

1. Joan Daemen and Vincent Rijmen, *The design of Rijndael: AES - the Advanced Encryption Standard*, Information Security and Cryptography, no. 17, Springer-Verlag, 2002.
2. United States National Institute of Standards and Technology, *Advanced Encryption Standard*, http://csrc.nist.gov/encryption/aes/, 2001.

The K Combinator as a Semantically Transparent Tagging Mechanism

Konrad Slind and Michael Norrish

School of Computing, University of Utah
Cambridge University Computer Laboratory

Abstract. The **K** combinator provides a semantically transparent tagging mechanism which is useful in various aspects of mechanizing higher order logic. Our examples include: numerals, normalization procedures, named hypotheses in goal-directed proof, and rewriting directives.

1 Introduction

Combinatory logic is based upon the two combinators **S** and **K**:

$$\mathbf{S}\ f\ g\ x = f\ x\ (g\ x)$$
$$\mathbf{K}\ x\ y = x$$

As is well-known, these two definitions are equivalent in power to Turing machines and the untyped lambda calculus. Combinators have also been used as the basis of abstract machines that implement functional programming languages, like Miranda [9]. General purpose computing machines based on combinators have even been realized as hardware. It is amazing that such a simple syntax is so powerful.

Our purpose is to expound another use of combinators, the **K** combinator in particular. On examination of **S** and **K**, one can (fancifully perhaps) see a split between **S**, which takes care of the functions, and **K**, which takes care of the data. Our interest is in representing particular *external* data in higher order logic. We will use instances of the **K** combinator in HOL as a tagging mechanism. The approach depends on the fact that an application

$$\mathbf{K}\ t_1\ t_2$$

has both the same type and the same meaning as t_1. We can put whatever well-formed term we wish in t_2. Thus if we want to somehow associate t_2 with t_1, we can transparently replace t_1 by **K** t_1 t_2. One use of this flexibility is to have t_2 be data that can be interpreted by external tools. Our examples show that the external tools can range from object-language syntax facilties like parsers and prettyprinters to proof support systems, to automated reasoners.

As might be expected, the **K** combinator is also used in functional programming. For example, in ML, with its left-to-right call-by-value evaluation strategy, the infix function **before** defined by

$$\mathbf{fun}\ (x\ \mathbf{before}\ y) = x$$

has the following behaviour: an expression M **before** N is evaluated by evaluating M, then N, and then the value of M is returned. Typically, evaluating N results in a side-effect (otherwise the use of **before** is pointless).

From our viewpoint, **K** is far more useful in higher order logic than in a programming language because one can both create and eliminate applications of **K** in logic, while only elimination is possible in a programming language.

In the following, we shall use some tags that are instances of the **I** combinator. This doesn't detract from our message, as we are thinking (not in any formal way) of **K** as a family of combinators:

$$\begin{aligned} &\mathbf{K}_0\ x = x \\ &\mathbf{K}_1\ x\ x_1 = x \\ &\mathbf{K}_2\ x\ x_1\ x_2 = x \\ &\quad\vdots \end{aligned}$$

Each $\mathbf{K}_n$ has type isomorphic to $\alpha \rightarrow \beta^n \rightarrow \alpha$. Only the first two members of this family are used in our examples.

1.1 Related Work

Kalvala has studied the application and implementation of tags [7]. The basic difference between her approach and ours is that she changes the underlying term structure to insert tags, while our tags arise from definitions and do not therefore require any changes to the kernel of the logic. Howe uses a tagging mechanism to attach types to the untyped terms of the Nuprl logic in [5]. Hutter [6] provides a *tour de force* of annotation uses, showing how they can be used to support such disparate applications as first order theorem proving heuristics (e.g. basic ordered paramodulation), window inference, rippling, and analogical reasoning. Like Kalvala, Hutter's approach requires altering the basic term structure.

2 Numerals

Our implementation of numerals for the natural numbers is similar to Harrison's in his HOL Light system. In contrast with earlier implementations of HOL, numerals are no longer members of an infinite set of constants. Instead they are *values*, constructed using three constants: 0, **NB1** and **NB2**. The two **NB** constants are defined

$$\begin{aligned} \mathbf{NB1}(x) &= 2x + 1 \\ \mathbf{NB2}(x) &= 2x + 2 \end{aligned}$$

Thus the number five is **NB1(NB2(0))**. This scheme has the advantage of unique representations for all numbers.

We use a $\mathbf{K}_0$ combinator, called **NUMERAL**, to tag all numerals explicitly at the outermost level. As Harrison notes in [4], this has the advantage that numerals are not sub-terms of other numerals. We have also found the tag idea useful in our implementation of arithmetic on these numerals. This implementation is based on Barras's implementation of "call-by-value" rewriting [1], to which we pass a variety of rewrite rules.

We begin by allowing addition to happen under the **NUMERAL** tag:

$$\mathbf{NUMERAL}(x) + \mathbf{NUMERAL}(y) = \mathbf{NUMERAL}(x + y)$$

A naïve implementation of addition could then use the following rules:

$$\begin{aligned} 0 + x &= x \\ x + 0 &= x \\ \mathbf{NB1}(x) + \mathbf{NB1}(y) &= \mathbf{NB2}(x + y) \\ \mathbf{NB1}(x) + \mathbf{NB2}(y) &= \mathbf{NB1}(\mathbf{SUC}(x + y)) \\ \mathbf{NB2}(x) + \mathbf{NB1}(y) &= \mathbf{NB1}(\mathbf{SUC}(x + y)) \\ \mathbf{NB2}(x) + \mathbf{NB2}(y) &= \mathbf{NB2}(\mathbf{SUC}(x + y)) \end{aligned}$$

Here the **SUC** ("successor") constant is being used liked a carry flag, to ripple along the rest of the computation. Unfortunately, in the absence of rules to pre-empt it, rewriting using the equations above won't emulate this rippling very well because all of the $x + y$ terms on the RHSs will be evaluated before the carry flag is used.

The first step of our solution is to not provide any rules for addition directly. Instead, all additions have to happen under a family of three **P** tags: $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$, where $\mathbf{P}_n(m)$ is defined to have the value $m + n$

(naturally, we won't be expanding $\mathbf{P}_n$ terms using this definition). $\mathbf{P}_0$ is thus another $\mathbf{K}_0$ tag. $\mathbf{P}_1$ corresponds to the use of **SUC** above. $\mathbf{P}_2$ is necessary because

$$\begin{aligned}\mathbf{P}_1(\mathbf{NB2}(x)+\mathbf{NB2}(y)) &= (2x+2)+(2y+2)+1\\ &= 2(x+y+2)+1\\ &= \mathbf{NB1}(\mathbf{P}_2(x+y))\end{aligned}$$

Luckily, there is no comparable need for a $\mathbf{P}_3$ flag when adding numbers under $\mathbf{P}_2$. We also change the rule for addition under **NUMERAL** to be

$$\mathbf{NUMERAL}(x)+\mathbf{NUMERAL}(y)=\mathbf{NUMERAL}(\mathbf{P}_0(x+y))$$

Next, we need rewrite rules to calculate the effect of $\mathbf{P}_1$ and $\mathbf{P}_2$ when applied to single arguments (the situation does not arise for $\mathbf{P}_0$):

$$\begin{aligned}\mathbf{P}_1(0) &= \mathbf{NB1}(0) & \mathbf{P}_2(0) &= \mathbf{NB2}(0)\\ \mathbf{P}_1(\mathbf{NB1}(x)) &= \mathbf{NB2}(x) & \mathbf{P}_2(\mathbf{NB1}(x)) &= \mathbf{NB1}(\mathbf{P}_1(x))\\ \mathbf{P}_1(\mathbf{NB2}(x)) &= \mathbf{NB1}(\mathbf{P}_1(x)) & \mathbf{P}_2(\mathbf{NB2}(x)) &= \mathbf{NB2}(\mathbf{P}_1(x))\end{aligned}$$

Finally, our set of equations for addition (omitting $x+y$ clauses when a clause for $y+x$ is already present) is then:

$$\begin{aligned}\mathbf{P}_0(0+x) &= x\\ \mathbf{P}_0(\mathbf{NB1}(x)+\mathbf{NB1}(y)) &= \mathbf{NB2}(\mathbf{P}_0(x+y))\\ \mathbf{P}_0(\mathbf{NB1}(x)+\mathbf{NB2}(y)) &= \mathbf{NB1}(\mathbf{P}_1(x+y))\\ \mathbf{P}_0(\mathbf{NB2}(x)+\mathbf{NB2}(y)) &= \mathbf{NB2}(\mathbf{P}_1(x+y))\end{aligned}$$

$$\begin{aligned}\mathbf{P}_1(0+x) &= \mathbf{P}_1(x)\\ \mathbf{P}_1(\mathbf{NB1}(x)+\mathbf{NB1}(y)) &= \mathbf{NB1}(\mathbf{P}_1(x+y))\\ \mathbf{P}_1(\mathbf{NB1}(x)+\mathbf{NB2}(y)) &= \mathbf{NB2}(\mathbf{P}_1(x+y))\\ \mathbf{P}_1(\mathbf{NB2}(x)+\mathbf{NB2}(y)) &= \mathbf{NB1}(\mathbf{P}_2(x+y))\end{aligned}$$

$$\begin{aligned}\mathbf{P}_2(0+x) &= \mathbf{P}_2(x)\\ \mathbf{P}_2(\mathbf{NB1}(x)+\mathbf{NB1}(y)) &= \mathbf{NB2}(\mathbf{P}_1(x+y))\\ \mathbf{P}_2(\mathbf{NB1}(x)+\mathbf{NB2}(y)) &= \mathbf{NB1}(\mathbf{P}_2(x+y))\\ \mathbf{P}_2(\mathbf{NB2}(x)+\mathbf{NB2}(y)) &= \mathbf{NB2}(\mathbf{P}_2(x+y))\end{aligned}$$

Our $\mathbf{P}_n$ tags can be seen as a specialised use of rewriting control, which we explore further below in Section 7.

3 Normalization

We have also used tags to implement a simple near-linear method for selecting and moving sub-terms to either end of a chain of arguments to an associative and commutative operator.

For example, when writing proof tools, it can be useful to have a particular conjunct at the front of the term, in a known position. If the input term is

$$P_1 \wedge P_2 \wedge \ldots Q \ldots \wedge P_n$$

and we wish to have Q at the front of the term, one approach to achieving this would be to prove the original term equal to a new one

$$Q \wedge (P_1 \wedge \ldots \wedge P_n)$$

General tools for doing such reordering proofs will necessarily take at least $O(n \log n)$ time however, and if the terms are very big this cost can be significant. Alternatively, one might write a bespoke "term-reorganiser" that carefully descended the term and did exactly the right sequence of transpositions to bring Q to the front.

With tags, we have another alternative again. Define a "marker" $\mathbf{K}_0$ tag, and descend the term to wrap it around Q. Then use the following theorems and HOL's general rewriter to bring $\mathsf{marker}(Q)$ to the front:

$$\begin{aligned} P \land \mathsf{marker}(Q) &= \mathsf{marker}(Q) \land P \\ P \land (\mathsf{marker}(Q) \land R) &= \mathsf{marker}(Q) \land (P \land R) \\ (\mathsf{marker}(Q) \land P) \land R &= \mathsf{marker}(Q) \land (P \land R) \end{aligned}$$

The final stage of the operation is to remove the marker wrapper from Q.

The use of the rewriter makes the implementation very simple, yet the efficiency will be close to linear (only a linear number of swaps will be made, but the rewriter may do some unnecessary work traversing other parts of the term looking for rewrites).

4 Constraint Tagging

The second author's implementation of Cooper's algorithm in HOL creates formulas that include terms of the form

$$\bigvee_{1}^{n} P(n)$$

for fixed n. Because n is fixed, such formulas could be expanded directly into n disjuncts, but it is more efficient to keep the disjuncts unexpanded so that later simplification can reduce the size of n. This would be the result of the so-called "δ-elimination" stage of the procedure, which might also replace n with an expression parameterised by variables that are in turn bound by other finite constraints.

HOL doesn't have an explicit parameterised disjunction operator, so we represent such formaulas with

$$\exists i.\ 1 \leq i \land i \leq n \land P(i)$$

For reasons of efficiency within the procedure, it is useful to be able to quickly locate the pair of constraints on variable i, and there is no guarantee that they will always be maintained at the front of the body of the quantification, as here. We wrap them inside a $\mathbf{K}_1$ tag, where the additional information is the variable i. This makes it easy to locate constraints over particular variables. The formula then becomes

$$\exists i.\ \mathbf{K}(1 \leq i \land i \leq n)\ i \land P(i)$$

5 Named hypotheses in proofs

Declarative proof interfaces have been a subject of recent interest in the interactive theorem proving community. In such a system [12, 3, 8, 13, 10], proofs are not given as a sequence of commands that alter a proof state—a *procedural proof* — but as a sequence of high-level assertions that closely follow the outline provided by a rigorous informal proof. Declarative proof systems offer readability and consequent advantages such as learnability and maintainability. In these systems, however, the original procedural proof interface—which is often appropriate in the heat of a proof is either unavailable, hidden, or deprecated.

This motivates the study of how combinations of declarative and procedural proof may be achieved. In fact, Harrison's work implemented declarative proof by procedural proof. However, he implemented a separate proof interface which one had to use to perform declarative proofs. In contrast, later work by Wiedijk [11] augmented the native procedural proof interface of HOL with declarative elements. One could mingle declarative steps with procedural steps, or indeed use only declarative steps. Importantly, only a single interface to proof was required. Unfortunately, Wiedijk's implementation was mostly aimed at showing how easy it is to build such an interface (it took about 40 lines of ML), and hasn't yet been developed into a user-friendly interface.

An important aspect of declarative proof is the attaching of names to hypotheses. In contrast, an important aspect of procedural (tactic) proof in the HOL88/90/98/4 implementations of HOL is that hypotheses in a goal are *not* named: they are a set. (The pros and cons of this have been extensively discussed in the HOL user community.) Thus any attempt to implement declarative proof in these systems will have to solve the problem of how to name and use hypotheses without having to rebuild the entire infrastructure of tactic proof.

Tags can be used to implement named hypotheses. We simply define a version of $\mathbf{K}_1$ as a logical constant

$$\mathbf{Named}\ (x : \mathit{bool})\ (y : \alpha) = x$$

Then an assumption A can be named n by **K**-expanding it to **Named** A n. Once a few simple tactics are written to access hypotheses by name, named hypotheses can exist in full harmony with unnamed hypotheses. This gives us a clean basis upon which to try to build declarative proof interfaces that co-exist with the existing interfaces.

An interesting subtlety, perhaps specific to HOL, is how to use named assumptions in tactics. For instance, the first-order model-elimination tactic **MESON_TAC** has type *thm list* $\rightarrow$ *tactic*. It uses the supplied theorems to prove a goal. Suppose we wish to provide a function **ASM** of type *string* $\rightarrow$ *thm* for fetching a named assumption from the hypotheses of a goal and making it a theorem, by assuming it. Thus we would be able to apply, *e.g.*, `MESON_TAC [ASM "foo"]` in order to use an assumption labelled with *foo* to prove a goal. However, the expression `ASM "foo"` must evaluate to a theorem, and the type of `ASM` forbids `ASM` from accessing the goal! (Using `top_goal` won't work.) Devious hackery is required. We manage to wriggle free of the conundrum by having `ASM` return an instance of reflexivity $\vdash foo = foo$, where *foo* is a variable. The preprocessing phase of **MESON_TAC** – which does have access to the goal – has been adjusted to find such trivial instances and turn them into accesses into the hypotheses of the goal.

6 Other operations on hypotheses

Two other applications of tagging support the abbreviation of subterms and the hiding of hypotheses.

1. In larger proofs, formulas with many repeated subterms can occur. To aid readability, abbreviation tactics have been written. Such a tactic will create a new assumption $v = M$ where M is the term to abbreviate, and v is a variable acting as its abbreviation, and replace all occurrences of M by v. However, such abbreviations don't work well with other tactics. For example, rewriting with the assumptions will re-expand any occurrences of v in the goal. For this reason, abbreviation tactics add the hypothesis in the reversed form $M = v$. However, this refinement is defeated by cleverer tactics that attempt to eliminate (by substituting throughout the goal) all equality hypotheses $v = M$ or $M = v$.[1] Such hypotheses often occur as the result of rewriting with injectivity theorems.
   The workaround is to introduce a $\mathbf{K}_0$ tag

   $$\mathbf{Abbrev}\ (x : \mathit{bool}) = x$$

   and then an abbreviation $v = M$ would be represented by the expansion $\mathbf{Abbrev}(v = M)$, and would be resistant to elimination by clever simplification tactics.
2. Another problem with larger proofs is extraneous hypotheses that clutter up the assumptions. They make the full goal hard to read, slowing interactive proof development. One way to deal with this is to eliminate them explicitly via a weakening tactic, but sometimes that is overly prescriptive. Another approach that may be useful would be to to introduce a $\mathbf{K}_0$ tag

   $$\mathbf{Hidden}\ (x : \mathit{bool}) = x$$

   and have the system prettyprinter omit assumptions of the form **Hidden** M.

[1] Note that the variable is restricted to occur on only one side of the equality, in order to preserve provability.

7 Rewriting directives

A basic functionality for a rewriter is to take a list of rewrite rules and apply them exhaustively to a term. Many other styles of rewriting are usually also required in proof assistants (rewriting with/without a background set, rewriting with/without the current assumptions, conditional rewriting, higher-order rewriting, *etc*), which leads to a large number of very closely related rewriting entrypoints, distinguished from each other by elaborate naming conventions, or multiple options, which may be confusing or hard to learn.

We can tackle some aspects of the complexity of this interface by using tags. Notice that the user may not want to treat all rules in R equally. For example, suppose one rule $\vdash r$ should be used twice, and the rest exhaustively. Our solution to this scenario requires some help from the meta-language. We define in HOL

$$\mathbf{BOUNDED}\ (b : \mathit{bool})\ (n : \alpha) = b$$

and in ML

```
datatype usage = UNBOUNDED
               | BOUNDED of int ref

fun Atmost th n = < ... create |- BOUNDED th n ... >
```

An invocation `Atmost` ($\vdash M$) i, where i is an integer, creates the theorem $\vdash$ **BOUNDED** $M\ n$, where n is a variable named i. This enables us to invoke the rewriter (which has to be altered, see below) with a list of theorems as follows:

```
REWRITE_TAC [Atmost r 2, ... ]
```

The rewriting engine pre-processes each rule to see if any are tagged with usage information. Those that *aren't* are paired with the ML value `UNBOUNDED`, and added to the set of rewrite rules. (Each element in the background set is paired with `UNBOUNDED`, reflecting the idea that such rules should be used exhaustively.) A rule that is tagged with a usage restriction is paired with the ML value `BOUNDED (ref n)`, where n is the supplied restriction. Once each rule has been mapped to being `UNBOUNDED` or `BOUNDED`, the rewriting process starts. When a rule is matched against the subterm being rewritten, it can be either unbounded, in which case the rewrite goes through, or bounded, in which case there is a check to see if the rule has been used up (*i.e.*, its reference cell holds '0'). If so, then the replacement doesn't happen. If not, the reference is decremented, and the replacement happens.

In HOL, only minor changes were made to the rewriting mechanism in order to have it process such tags. What is pleasant is that the code is completely backwards compatible: existing applications of the rewriter in tactic scripts do not need to be changed.

As future work, we wish to add in a rewriting directive for conditional rewriting. There are essentially two ways to implement conditional rewriting in goal-directed proof. The standard approach demands that all conditions be proved before replacement takes place. This is usually the desired behaviour, but occasionally the proof of the conditions fails, and it can become an awkward business to get the rewrite to happen. The alternate approach *assumes* the conditions, thus ensuring that the rewrite happens, leaving the conditions to be polished off later. Accomodating the two styles would seem to require multiple entrypoints, but we envision having a **Force** tag that would signal which manner of rewriting should be used on a conditional rewrite rule. One interesting outcome is that it may be possible to perform induction proofs via forced and bounded higher-order rewriting (since an induction theorem has the form of a conditional higher-order rewrite).

8 Conclusions and Future Work

We have seen how some common implementation issues in higher-order logic theorem provers can be handled with a notion of transparent tagging. We hope that giving a name to a common practice will encourage others to come forward with their own tales of using **K** to support proof.

The device is not without its drawbacks. For example, it seems best to restrict the scope of a tag's usage: tags are best when only their implementor is aware of their existence. Thus, eliminating a tag as soon as possible seems to be good practice. Otherwise, unrelated proof tools may need to know about the tags used by each other, making for a development nightmare. It is true that some tags, like those for numerals, do persist; however, they don't seem to cause much trouble (perhaps that is because they represent constants).

Another (related) worry is nesting of tags. In that case, the semantics of **K** mean that no confusion of meaning is possible, but confusion of proof tools may certainly happen. For example, what if an assumption is named twice? With different names? Tag creation and elimination could be made idempotent, but the issue remains, especially when tags supporting different proof tools overlap.

A further limitation is that tags need to be well-typed terms. If that is a problem, one can use strings, uninterpreted constants, or the names of free variables in order to provide tags that can be externally interpreted. If, for example, one wished to attach hyperlinks—in the logic—to theorems, a string tag

$$\mathbf{URL}\ (t : \mathit{bool})\ (s : \mathit{string}) = t$$

might be a possibility.

We have seen how tagged terms may be implemented: what about tagged types? To follow our initial insight, we need types that act like **K**. This may correspond to so-called *phantom types*, in which superfluous type variables are used to enforce extra invariants via type inference. The paper [2] provides a range of applications of phantom types in interfacing C to ML, including the use of type inference to enforce array bound constraints. Thus, like tags, phantom types are useful for building interfaces between a type theory and the outside world. It may be possible to create phantom types in HOL as well.

References

1. Bruno Barras, *Proving and computing in HOL*, Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings (Mark Aagaard and John Harrison, eds.), Lecture Notes in Computer Science, vol. 1869, Springer, 2000, pp. 17–37.
2. Matthias Blume, *No longer foreign: Teaching an ML compiler to speak C "natively"*, Electronic Notes in Theoretical Computer Science **59** (2001), no. 1, 1–16, http://www.elsevier.nl/locates/entcs/volume59.html.
3. John Harrison, *A Mizar mode for HOL*, Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96 (Turku, Finland), Lecture Notes in Computer Science, no. 1125, Springer-Verlag, 1996, pp. 203–220.
4. ______, *Theorem proving with the real numbers*, CPHC/BCS Distinguished Dissertations, Springer, 1998.
5. Doug Howe, *A type annotation scheme for Nuprl*, Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98 (Canberra, Australia), Lecture Notes in Computer Science, no. 1479, Springer-Verlag, October 1998, pp. 207–224.
6. Dieter Hutter, *Annotated reasoning*, Annals of Mathematics and Artificial Intelligence **29** (2000), 183–222.
7. Sara Kalvala, *Annotations in formal specifications and proofs*, Formal Methods in Systems Design **5** (1994), no. 1/2, 119–144.
8. Donald Syme, *Three tactic theorem proving*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999, pp. 203–220.
9. David A. Turner, *Miranda: A non-strict functional language with polymorphic types*, Proceedings of the 2nd Conference on Functional programming languages and computer architecture (Nancy, France) (J.-P. Jouannaud, ed.), Lecture Notes in Computer Science, vol. 201, Springer-Verlag, September 1985, pp. 1–16.
10. Markus Wenzel, *Isar — a generic interpretative approach to readable formal proof documents*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999, pp. 167–185.
11. Freek Wiedijk, *Mizar: An impression*, available at `http://www.cs.kun.nl/~freek/mizar/mizarintro.ps.gz`, 1999.
12. ______, *Mizar Light for HOL Light*, Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001 (Edinburgh), Lecture Notes in Computer Science, no. 2152, Springer-Verlag, 2001, pp. 378–393.
13. Vincent Zammit, *On the implementation of an extensible declarative proof language*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999, pp. 185–202.

Real Numbers in Real Applications
Abstract

John Harrison[1]

Intel Corporation
johnh@ichips.intel.com

Abstract. The formalization of mathematics in theorem provers and proof checkers, including continuous mathematics such as real analysis, is sometimes undertaken purely for intellectual interest. For example, the Mizar Mathematical Library includes a large number of analytical theorems. But a surprising phenomenon is how useful non-trivial mathematics can be in verification applications.
One might guess that for verification of concrete floating-point algorithms, only the most basic "algebraic" properties of reals and simple combination formulas for transcendental functions would be needed. But we will draw on our own experience to show that this is not so, and one needs a surprising amount of pure mathematics. Thus we can, should we so wish, justify the formalized development of much apparently "impractical" pure mathematics even in crudely utilitarian terms.

A PVS Service for MathWeb

A. A. Adams[1], A. Franke[2], and J. Zimmer[2]*

[1] School of Systems Engineering
The University of Reading
A.A.Adams@Rdg.ac.uk

[2] AGS, Universität des Saarlandes
{jzimmer,afranke}@ags.uni-sb.de

Abstract. MathWeb is a system which allows mathematical software programs to intercommunicate. The aim is to allow manual or automatic queries from, say, a higher order theorem prover to a first order theorem prover or from a computer algebra system to a theorem prover. We present an implementation of a basic PVS service in MathWeb. The service offers a black box which takes PVS-syntax conjectures and interprets the resulting PVS output as to whether the proof attempt succeeded or not. The main implementation allows access to the PVS Real Analysis Library and defaults to running the *grind strategy* ("strategy" is the PVS term for what is often called a *tactic* in other systems) on the submitted conjecture. Customisation to access other libraries and strategies is possible, and we present an instantiation of this to access Gottliebsen's continuity checker for the Real Analysis Library. We also give an overview of the difficulties of accessing PVS is this way with suggestions for abstracting the core prover away from the existing EMACS interface.

1 Introduction

In this paper we present an initial implementation of a PVS-MathWeb-Interface [PVS-MWI] and details of the proposal for a full implementation of such an interface. PVS is a higher order theorem proving system designed for interactive use primarily in formal methods development. Theory developments in real analysis, however, have made it a useful tool for work in supporting computer mathematical assistance but the need for theorem proving technology in this area lends itself more to a black box theorem proving system than an interactive theory development platform. In addition, interoperability with various other systems is important in this application domain. The MathWeb software bus is a useful broker and integration architecture for such systems and as such is an obvious platform into which to link the PVS system, and in particular the real analysis capabilities of Gottliebsen's development [8, 9].

We begin with some background information on MathWeb in the next section and then proceed with background on PVS and the Real Analysis Library. Next we present some of the problems of running PVS, designed to be an interactive theory development system, as a black box prover in section 4. In sections 5 and 6 we first present the proposed PVS-MWI and then the prototype implementation. We finish by considering the ramifications of this work and future directions for such a development in section 7.

2 Background: MathWeb

The MathWeb software bus has been developed, primarily at the University des Saarlandes, to act as an intermediary between various pieces of software which perform symbolic calculations of some form. The original aim was to allow theorem provers [TP] and computer algebra systems [CAS] to interact. Previously, a number of pairings of such systems had been connected but this necessitated working out the exact details of external communication for each system in turn. The advantage of a system like the MathWeb software bus is that any system connected to it can be connected to any other system without worrying about operating system level details of their communication. In particular, any system which has a straightforward command-line interface can be fairly easily connected to MathWeb.

The details of connecting systems together requires more than simply the operating system level of connections, of course. Problems which must still be addressed for each system include:

* This work was supported by the EU Grant Calculemus HPRN-CT-2000-00102.

- Translation between the object languages of the TP or CAS.
- Control of the processing occurring in each system.
- Synchronisation of a mutual symbolic calculation effort.

Use of the OZ language (using the Mozart implementation: see [12]) allows MathWeb to provide a solid platform in which these problems may be addressed while at the same time providing a fully transparent system of communication on a local host, a LAN or fully distributed over the internet.

3 Background: PVS Real Analysis Library

First we consider the uses of formalisations of real analysis in general and then give details of the formalisation in PVS.

3.1 Formalisation of Real Analysis

The formalisation of real analysis and related continuous mathematics topics, is currently an area of great interest. Most of the major higher order logic systems currently available have one or more efforts underway to develop a formalisation of real numbers, transcendental functions, or similar. This development stems from a number of different application areas, including:

- Formal Methods used to support the development of air traffic control systems. This requires a formalisation of geometric aspects of air flight, which requires trigonometric functions as part of the library of underlying concepts: [3]. The development in PVS of a Real Analysis Library has been useful in such work.
- Formal proof that a hardware implementation of the IEEE floating point operations require an underlying concept of the operations over the reals to be present to allow formal proof that the rounding operation is not introducing errors: [10]. This led to the development of a Real Analysis Library in HOL-Light, much of which has been ported to HOL-98.

Development of theories of real analysis, complex analysis and related topics are available or under current development in PVS [5,8], HOL (HOL-Light and 98) [10], Isabelle (Isabelle/NSA) [6], ACL2 (actually a variant called ACL2(r)) [7] and Coq.

3.2 The PVS Real Analysis Library

The particular work we are interested in here is the development of a Real Analysis Library for PVS. The initial work on this library was done by Dutertre [5]. PVS already included a base type of `real` as an axiomatised sub-type of `number`. Dutertre developed various parts of fairly abstract real analysis. That development started with a theory of sequences of reals (PVS has polymorphic sequences as part of the base system), developed explicit convergence criteria for sequences and then functions on the reals, and finally defined abstract notions of continuity and differentiation. With some changes to make concepts such as continuity and differentiation more useful for concrete functions, Gottliebsen [8] extended this library with further work on real functions, including the definition of infinite sequences and power series. The definition of power series allowed an analytic definition of a number of the transcendental functions, including ln, exp, sin, cos an tan. The properties of these functions were developed as a lemma data base to the point where they were sufficiently characterised that further lemmas can generally be proved without reference to the underlying power series. The definition of any new transcendental function would require a power series style of definition, however, and a similar background of properties would need to be added for the new function to be useful.

3.3 Automated Proving Using the PVS Real Analysis Library

The PVS Real Analysis Library was primarily developed to support research in improving the capabilities of Computer Algebra Systems [CAS] such as Maple and Mathematica. As such, the aim was to couple a CAS and PVS together to produce a stronger (i.e. capable of producing correct answers in more circumstances). The original project looked primarily at the contents of the computational mathematics being considered by each system rather than at practical measures for coupling such system together. The utility of PVS in these circumstances was as a black box which could handle logical side conditions on various computations. In particular the inability of CAS to show satisfiability or unsatifiability of sets of constraints on real-valued parameters with equalities and inequalities involving transcendental functions, causes great problems in the area of definite integration (amongst many others). In addition, the same problem area of definite integration leads to a need for a good continuity checker for parametric real functions, another area in which CAS are particularly weak.

The automatic proving features of PVS were originally designed to perform "routine" tasks during interactive proving. The scope of these "routine" tasks has gradually increased to the point where these automatic routines can now handle fairly complicated problems without any user intervention. PVS' automated proof procedures build up in a hierarchy of rewriting, ground term evaluation and definitional unfolding. The top-level generic automated procedure which we are interested in is *grind*. This strategy has a large number of optional arguments which control the operation of the underlying procedures to guide the proof search. While PVS has a strategy language in which specialised strategies may be developed, it is often more useful to use this language to define a special-purpose strategy which simply calls grind with appropriate arguments. The continuity checker developed by Gottliebsen [9] uses this method. Investigations continue into the best parameter settings for grind to cope with satisfiability problems.

Recent work at the NASA Langley Research Centre and ICASE has led to some interesting developments in special purpose tools for automated proof of formulae involving real numbers. See [4] for details.

4 Running PVS as a Black Box

As mentioned above, PVS was designed very much as an interactive system. The developers of the system, who are also themselves one of the primary user groups, use the system in this way. Only a few small projects have tried to use PVS as a back end system with a different interface and as such, the integration of PVS as an automated system available via MathWeb produces some interesting implementations problems. This section will look at the structure of PVS and explain where the MathWeb interface should sit in the system.

4.1 The Structure of the PVS System

The main user interface for PVS is written in Emacs (also compatible with XEmacs). The core engine is written in Allegro Common Lisp and distributed as a run-time library only. The PVS development team are considering options for releasing the source code of the core engine, but it was not available at the time this project was undertaken, although Owre of SRI has been particularly helpful in identifying the hooks between the core engine and the Emacs interface. On first evaluation, the system is deceptively simple, and may be considered as shown in figure 1.

Looking closely at the details, however, we see that there is not such a strong divide between the operations of the Emacs Lisp Interface and the Allegro CL core. The Allegro CL Core writes information directly to temporary files, for instance, which the Emacs Lisp Interface then copies to the permanent storage position. Message passing is also not quite as simple as we would wish: the Allegro CL Core prints out large amounts of text delimited by message markers such as

```
:pvs-msg          :end-pvs-msg
```

which are parsed by the Emacs Lisp Interface and displayed (or not) in the correct manner (via the interaction window, via the message bar or written to Emacs buffers). In addition, as well as input from the user, the Emacs Lisp interface controls the timing with the core engine by sending appropriate signals to let the core engine know that it is ready to process the next part of its output. The Allegro CL Core waits for these messages before it proceeds with the proof attempt.

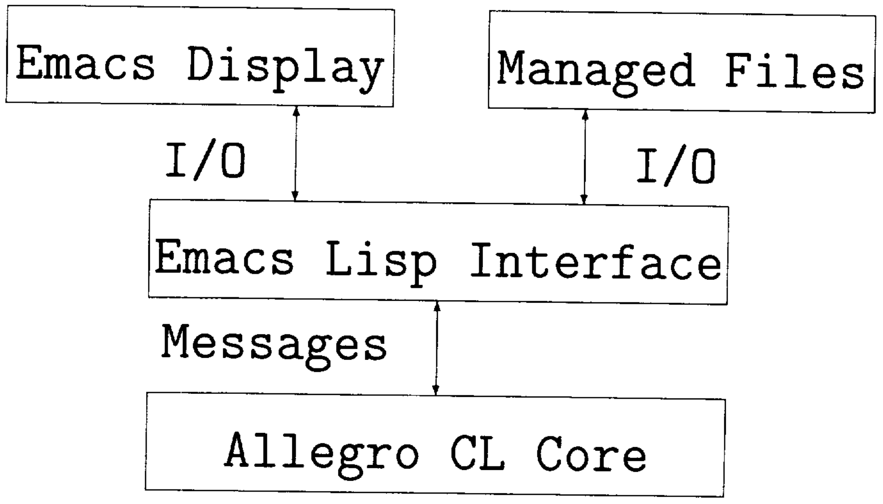


Fig. 1. A Naive View of the PVS System

4.2 Design of the PVS-MWI

The eventual aim of the PVS MathWeb service is also to be able to offer an interactive proof session. This would comprise simply the ability to perform interactive proofs and to parse the produced proof script. The theory management aspects, such as adding new theorems to a library, are not expected to be supported in this manner. For the moment, and as a first step, however, we aimed at allowing the MathWeb server to send a problem in PVS syntax to a PVS core engine running as a stand-alone process, and to interpret the results as success or failure. This initial project aims to develop an understanding of the issues and to identify the aspects of PVS that are suitable for such an interface, and those which require revision to ensure a robust system.

Given that most PVS development and usage is expected to continue via the good interactive interface already available, the ability to talk to an external server such as MathWeb is seen as a side goal of the main development, but one which should not interfere with the primary development path. Thus we present a design here which would allow our aims of an automatic or interactive proof session to be sent to PVS, and the resulting proof script to be interpreted. Then we will present the aspects of PVS which are relevant to this design, and finally present the existing state of our interface, which achieves the basic aim of allowing automatic proof attempts in PVS with the very limited result of checking for success or failure. The design presented here is to work with the PVS system in its current form. See section 7 for a discussion of changes to the core engine which would allow a cleaner PVS-MWI interface without degrading with the existing interface.

4.3 Existing PVS System

Figure 2 shows a more detailed breakdown of the actions of the PVS Allegro CL Core and the Emacs Lisp Interface

The Interface reads files from the permanent file store, which provide persistence across sessions for theory development, proof scripts and other support mechanisms. It also controls information flows into the Emacs buffers which hold the raw output of the Core Engine, the interactive session with the user and theory/proof files currently under development. The Core Engine outputs messages to the Interface (interpreted via the raw output buffer), and also writes proof scripts directly to the temporary file store. Most importantly of all, the Interface sends commands directly to the Core Engine. These commands include start-up routines, the loading of theory files and libraries, and proof commands.

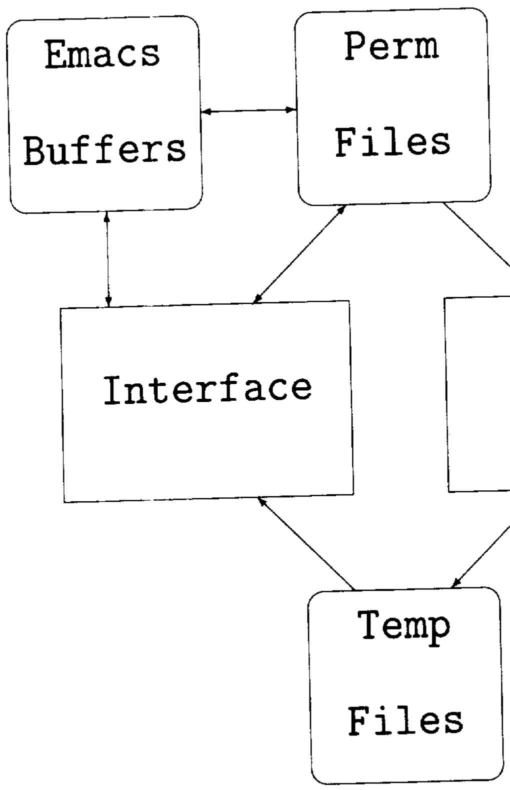


Fig. 2. A Model of PVS Operation

4.4 Automated Proving in PVS

Before we consider our proposed and initial implemented interfaces to PVS in MathWeb, we must present an unusual aspect of PVS, which will inform our discussion later. PVS is not strictly an LCF-style system. Proofs are not necessarily produced by a sequence of atomic logical operations. Instead, a number of the PVS strategies are written as compiled programs which are linked in to the system. This was a design decision to enhance the usability of PVS by speeding up the proof process of particular heavily used strategies such as *grind*. Since PVS does not produce proof objects as such, but only proof scripts, this does not lead to problems generally. In order to allow greater assurance, however, PVS has a facility whereby these external proof procedures can be called as transparent strategies instead. These transparent versions expand to individual logical operations, and it is expected that the actions of the transparent and opaque versions of the strategies will have the same effects given the same inputs. The thinking behind this is that while proceeding with a development the user is interested in quick results from the decision procedures. Once a development has been completed the user may then gain extra confidence (or insight into the resulting proofs) by re-proving the theorems to expand the proof procedures. This will become important when we consider the output of a call to PVS from an external system, which may or may not require a proof object as well as an indication of success or failure.

5 Proposed PVS-MWI

To produce a complete MathWeb service for PVS we wish to replace the interface segment with a new facility. This new PVS-MWI need not duplicate the entire functionality of the existing interface. For instance, it need not write to permanent PVS files. Nor is there any requirement to write information to Emacs buffers. However, it will need to read from the permanent files of the PVS installation, and from the temporary files written by the Core Engine. The PVS-MWI also needs to communicate directly with the Core Engine.

5.1 The Automatic PVS Proof Service

As mentioned above, we are not aiming to provide a replacement for the existing Emacs interface. Instead we aim to provide a PVS server which will allow proofs to be performed either automatically or interactively over a remote connection, almost certainly via another program. While much of the Emacs interface deals with developing theories and associating proofs with their formulae etc., we are not concerned with such issues. However, there is one issue regarding theories that we must address: the context in which a formula to be proved is valid. It is unlikely that many users will wish to use simply the base PVS logic and initial developments (the "prelude" theories) only. It is more likely that a particular theory context will be required, such as the real analysis library, the context offered by the current version of the PVS-MWI.

Since there is a substantial time-lag in loading large theories such as the Real Analysis Library, it is expected that each service offered over MathWeb will actually be an image with the appropriate theory pre-loaded. Thus, instead of simply asking for a PVS service, the client program would have to specify PVS+*Lib*, where *Lib* might be *Reals*. To ensure efficient communication, the ideal interface should also indicate the type of information expected as a return. There would be no point in returning a proof to a computer algebra system for instance, whereas Ωmega requires prover services to return an appropriate proof object. We can classify the modes of operation required of the interface as shown in table 1 below.

Query Mode	Responses
Prove Positive, No Object	True/Unknown
Prove Positive&Negative, No Object	True/Unknown/False
Prove Positive, Object	True/Unknown&Proof
Prove Positive&Negative, Object	True/Unknown/False&Proof

Table 1. Modes of PVS Interaction

As mentioned above, the modes which require the return of a proof object (in the case of PVS an expanded proof script is the best that is available) will require a slightly different form of the strategy command passed to the PVS core engine. Thus the command in a client which passes a conjecture for proof to PVS should have five arguments:

1. the conjecture (a formula in PVS syntax or another syntax which the PVS-MWI can convert into PVS syntax);
   No default value;
2. a flag indicating whether an attempt to prove the formula True is required or an attempt to prove the formula true or false: PVS does not return "False", it simply returns True or fails to complete a proof;
   Default value: True Only;
3. the name of a strategy to call to attempt to prove the conjecture;
   No default value;
4. the name of the theory in which context the proof of the conjecture should be attempted;
   Default value: prelude;
5. a flag to indicate whether a proof object should be returned;
   Default value: False.

Should a PVS-MWI contain a translation mechanism from, say, OpenMath notation into PVS notation then a flag indicating the syntax used in stating the conjecture may be needed. Likewise should the interface be capable of returning proof objects in different formats then an extra argument indicating the required return format would be useful.

Note that the lack of a default value for the strategy to be applied allows for an "empty" strategy to be passed in to indicate that a user wishes to perform an interactive proof.

The current interface must perform a fair amount of processing of PVS I/O at start-up and during each proof attempt. In the long run much of this work should be ameliorated with access to extend the PVS Allegro Common Lisp Core with appropriate flags indicating the status of PVS in a "black box" mode.

It should be noted that there are a number of non-standard cases to be considered for a full-featured automatic proof service:

- A conjecture may be submitted that is syntactically or type incorrect.
- PVS does not attempt to prove a conjecture false, so any false conjecture submitted will simply lead to a failure of proof, indistinguishable from a case where the proof strategy is not strong enough to prove the conjecture.

6 Current Version of the PVS-MWI

In this section we will describe the current minimal implementation of the PVS-MWI. This is a working service, available for installation as part of MathWeb, but includes only parts of the full service described in the previous section. See the later section 7 for details of ongoing development in this project.

6.1 Starting PVS

Originally, running PVS in a shell required the execution of the Allegro Common Lisp image directly, rather than invoking the distributed shell script which normally starts PVS. Since then Owre has added a switch to the shell script which runs PVS as a stand-alone program: `pvs -raw`. This still only starts the core engine of PVS, however, and there are various steps which must be then taken to put the system into a usable mode as a proving tool. The first of these has been made obsolete by the most recent release of PVS (2.4). We are in the process of updating the PVS-MWI to take account of this release.

1. Load the latest patch files (if patch files are present).
2. Change to the "Package" PVS (see the Common Lisp documentation [13] for a description of packages).
3. Change the working directory in which PVS operates (in interactive sessions this is called the context).
4. Run a simple test proof of "1=1" using "(grind)" in the context of the Real Analysis Library. This pre-loads the entire library into the current image.

6.2 Successful Proof Process

The PVS Core Engine function `prove-formula-decl` is the function called to start the proof process. The arguments of this function include the conjecture for which proof is to be attempted, the PVS theory which forms the context of the conjecture and proof attempt, and a strategy which is to be applied.

At specific points in the proof attempt process, the PVS Core Engine outputs a message indicating to the Emacs interface that it is in a particular state and ready to proceed with the next stage of the proof. At each of these points, the PVS Core Engine expects a token from the Emacs interface to indicate that it is ready to proceed with the next phase of the proof. This interaction is due to the requirements of the Emacs interface to display messages, update the interaction buffer, and copy text to and from temporary file storage. The sequence of interactions is shown below. The text in `courier typeface` is the output from the PVS Core Engine (... indicates other lines appearing first). The conjecture being proved is `NOT (0=1)`, by using the strategy `(grind)` in the context of the Real Analysis Library.

```
...
:pvs-msg Formula typechecked :end-pvs-msg
:pvs-eval (setq pvs-in-checker t) :end-pvs-eval
```

This shows that the Core Engine has completed type checking of the conjecture. Note that this does not necessarily mean that type checking has succeeded, simply that it hasn't failed. See section 7 for a discussion of this. The next line indicates that the Core Engine is now entering proof check mode. Once a token ("t" is appropriate) has been sent to the Core Engine, we get the following:

```
test :
{-1}  (0 = 1)
  |-------
Rerunning step: (GRIND)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
:pvs-eval (setq pvs-in-checker nil) :end-pvs-eval
```

The name of the conjecture being proved (important when the interface is maintaining a record of a theory and linking a proof to lines of a theory file) is `test` (the specification of the conjecture in the argument of `prove-formula-decl` is required is have a name). The initial form of the sequent in the proof is printed and the steps to be automatically attempted in the roof are shown (in this case simply the single step `(GRIND)`). The progress of the strategy is reported (`Trying...`) and then success is indicated with `Q.E.D.` followed by the fact that the Core Engine is dropping out of proof checker mode. Again, a token is sent to the Core Engine, resulting in the following output:

```
:pvs-eval (pvs-ready) :end-pvs-eval
```

A final token submitted to the Core Engine returns "T". This is the evaluation result of `prove-formula-decl`, but is not an indication of the success or failure of the proof attempt, simply a place-holder return value.

6.3 Unsuccessful Proof Process

The above sequence shows what happens when PVS is presented with a conjecture which is provable by the strategy requested. This is not always the case, however, so we must consider how the Core Engine acts when presented with a conjecture unprovable by the strategy. This can be because the strategy is not strong enough to prove a true conjecture or because the conjecture is false. Note for our initial implementation of a PVS-MWI we have not implemented a recovery scheme for cases where an ill-formed conjecture is submitted (either with a syntax error or an identified type-checking error). Nor has recovery from an apparent "infinite loop" been implemented by sending a break signal and recovering from the resulting break level of the Allegro Common Lisp session in which the Core Engine runs.

To demonstrate an unsuccessful proof attempt we will present the interaction which occurs when the conjecture `0=1` is presented to the `(grind)` strategy:

```
...
:pvs-msg Formula typechecked :end-pvs-msg
:pvs-eval (setq pvs-in-checker t) :end-pvs-eval
```

An identical start to the previous sequence, with the formula correctly type checked and the Core Engine reporting that it is now in proof checker mode.

```
test :
  |-------
{1}   (0 = 1)
Rerunning step: (GRIND)
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
test :
  |-------
[1]   (0 = 1)
***Warning: Fewer subproofs (0) than subgoals (1)
No change on: (SKIP)
test :
  |-------
[1]   (0 = 1)
Postponing test.
```

```
test :
  |-------
[1]   (0 = 1)
:pvs-eval (setq pvs-in-checker nil) :end-pvs-eval
```

Having received a token, the (grind) strategy this time fails to prove the conjecture. The critical point here is actually the *lack* of an appearance of Q.E.D. being the easiest method of detecting failure. Another token is then sent to the Core Engine, which has indicated that it has dropped out of proof checker mode:

```
Would you like the partial proof to be saved?
(***Old proof will be overwritten.***)
(Yes or No)
```

Now we see the unwanted interaction designed for the full Emacs interface coming into play, asking the user for instructions with regard to the partial proof developed in attempting to prove the conjecture. The Core Engine actually does very little depending on the answer here. The primary operation is carried out by the Emacs Interface in copying the temporary proof file into the appropriate part of the proof file underlying the current theory. On sending "yes" the Core Engine returns:

```
Use M-x revert-proof to revert to previous proof.
:pvs-eval (pvs-ready) :end-pvs-eval
```

For the purposes of a replacement interface, however, "no" is a more appropriate response, to which the Core Engine gives:

```
:pvs-eval (pvs-ready) :end-pvs-eval
```

After either of these, a further token is required to reset the Core Engine to it's top-level read-eval-print loop, with the final response of:

```
("" (GRIND))
```

that is, the partial proof.

6.4 The Implemented Basic PVS-MWI

Thus the basic PVS-MWI that has been implemented has the following features:

- Only the actual conjecture (in PVS syntax) is a required argument. To this is added the "test: lemma " text. Default values of the Real Analysis Library for the context and (grind) for the strategy are included.
- Three main functions are provided:

Function	Description
prove	Simply attempts to prove the conjecture and returns "True" if successful. If this fails it returns "I Don't Know!"
provetf	Attempts to prove the conjecture True and returns "True" if successful. If this fails then it attempts to prove the negation of the conjecture and returns "False" if this succeeds. If both fails it returns "I Don't Know!"
provects	This assumes that the conjecture is of the form of continuity of a real-valued function. It calls Gottliebsen's continuity-checker (cts) and returns "True" if this succeeds and "I Don't Know" otherwise. Proof of discontinuity has not been implemented so proof of the negation here is not supported.

- In all cases where a proof attempt fails, the PVS-MWI answers "no" to the question posed about recording the partial proof and correctly resets the Core Engine to the top-level prompt.

7 Conclusions and Future Work

The existing interface is simply a prototype and requires a fair amount of further work to be truly useful. Nevertheless, the exercise has proved very useful in a number of ways, which we will consider here, before addressing the future direction of this work.

Taking a system such as PVS which was designed as an interactive theory development platform with a specific Emacs interface, and allowing its use as a black box back end theorem prover has proved more complicated than might be initially expected. However, the areas where optional settings may be added in to make this an easier proposition have now been identified and development of PVS in this direction should not prove difficult.

Related work on system specific interface such as the Maple-PVS link [1] should also benefit from this exploration of the PVS system and developments in this area.

Various aspects of the MathWeb software bus have been tested and occasionally broken during this development. This paper has not focussed on such details as race conditions between the interface and the PVS Core Engine; zombie PVS processes caused by failures of the broker architecture; and similar. Nevertheless the development of the MathWeb architecture has undoubtedly benefitted from including PVS in its family as proof system server. Developments in using other higher order systems as servers in MathWeb will be easier following the lessons learned here. Specification of a generic black box automated theorem proving service derived from the PVS service is one concrete outcome for the MathWeb system from the prototype PVS-MWI.

7.1 Exploration of PVS as a Back End System

As mentioned many times above, PVS was primarily developed as a theory development platform and it will certainly continue to be used in such a fashion. Indeed, further enhancement of the capabilities as a black box system require a good theory development platform to be available. However, identifying the areas of the existing PVS system where the theory development platform is unnecessarily embedded in the core engine will prove a useful exercise in informing future development of PVS. Once a theory has been developed it is quite often useful to allow black box use of automated strategies in the context of that theory and the work shown here will hopefully allow further development of the PVS Core Engine to support this need. Availability of PVS theories as black box systems should also stimulate development of more and more complicated theory systems in addition, benefitting the PVS community as a whole.

7.2 Implementation of the Full PVS-MWI

The main task of the development of a full PVS-MWI would be the development of an interactive proof ability. Extension of the existing prototype to cover black box proving as shown in section 5 should be relatively straightforward, providing changes to the PVS Core Engine as described below are undertaken.

Some rationalisation of the existing system would be required, most specifically a separation from the current dependence on the Real Analysis Library as the default context. An early decision would need to be made as to whether a generic PVS prover might be offered or whether specific provers offering a single context would be better. Each has its advantages, and the decision would also be informed by the changes that might be made to the Core Engine of PVS.

It had been thought that automated proof checking using `prove-formula-decl` generated separate type checking conditions (tccs). On communication with PVS developers, however, it turns out that tccs generated when performing a proof with this function are folded into the goal, so that whenever "Q.E.D." is generated, one can be sure that any tccs have also been verified. However, this behaviour may be the cause of occasional infinite loops during the automated proof attempts. A more sophisticated approach to using PVS as an automated back end proving system might do something more intelligent with the tccs.

7.3 Proposed Developments of the PVS Core Engine

We list here suggested amendments to the Core Engine which are feasible without altering the existing Emacs interface code substantially.

- Switching off the printing of most of the messages.
- Not requiring a token passed in at the various points noted above.
- Ignoring the "partial proof" to be saved or not when a failure of proof occurs.
- A robust timeout setting returning a failure of the proof attempt after a certain number of CPU cycles.
- Defining a new proof function which returns a `T` or `NIL` as well as the proof if successful.

In addition, Allegro Common Lisp run-time images of the Core Engine with various libraries pre-loaded would improve the efficiency of start-up of a PVS-MWI which offered services in those contexts.

References

1. A. A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer Algebra meets Automated Theorem Proving: Integrating Maple and PVS. In Boulton and Jackson [2], pages 27–42.
2. R. J. Boulton and P. B. Jackson, editors. *Theorem Proving in Higher Order Logics: 14th International Conference*. Springer-Verlag LNCS 2152, 2001.
3. V. Carreño and C. Muñoz. Aircraft Trajectory Modeling and Alerting Algorithm Verification. In Harrison and Aagaard [11], pages 90–105.
4. B. Di Vito and C. Muñoz. New PVS Prover Strategies for Real Arithmetic. http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS2-library/pvslib.html.
5. B. Dutertre. Elements of Mathematical Analysis in PVS. In von Wright et al. [14], pages 141–156.
6. J. D. Fleuriot and L. C. Paulson. Mechanizing Nonstandard Real Analysis. *LMS Journal of Computation and Mathematics*, 3:140–190, 2000.
7. R. A. Gamboa and M. Kaufmann. Non-Standard Analysis in ACL2. *Journal of Automated Reasoning*, 27(4):323–351, Nov 2001.
8. H. Gottliebsen. Transcendental Functions and Continuity Checking in PVS. In Harrison and Aagaard [11], pages 198–215.
9. H. Gottliebsen. *Automated Theorem Proving for Mathematics: Real Analysis in PVS*. PhD, School of Computer Science, University of St Andrews, 2001.
10. J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.
11. J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*. Springer-Verlag LNAI 1869, 2000.
12. The Mozart Implementation of the OZ Programming Environment. http://www.mozart-oz.org/.
13. G. L. Steele Jr. *Common Lisp: The Language, Second Edition*. Digital Press (Bedford, MA), 1990.
14. J. von Wright, J. Grundy, and J. Harrison, editors. *Theorem Proving in Higher Order Logics: 9th International Conference*. Springer-Verlag LNCS 1125, 1996.

Formalizing Real Calculus in Coq

Luís Cruz-Filipe*

Department of Computer Science, University of Nijmegen, The Netherlands
Center for Logic and Computation, IST, UTL, Portugal
`lcf@cs.kun.nl`
`http://www.cs.kun.nl/~lcf`

Abstract. We have finished a constructive formalization in the theorem prover Coq of the Fundamental Theorem of Calculus, which states that differentiation and integration are inverse processes. This formalization is built upon the library of constructive algebra created in the FTA (Fundamental Theorem of Algebra) project, which is extended with results about the real numbers, namely about (power) series.
Two important issues that arose in this formalization and which will be discussed in this paper are partial functions (different ways of dealing with this concept and the advantages of each different approach) and the high level tactics that were developed in parallel with the formalization (which automate several routine procedures involving results about real-valued functions).

1 Introduction

In this paper we show how a significant part of real analysis can be formalized in Coq. We deal with differentiation and integration, proving the Fundamental Theorem of Calculus (which states that differentiation and integration are in some sense inverse processes) and Taylor's Theorem (which allows us to express a function in terms of its derivatives while giving an estimate for the error), as well as defining some standard constructions such as function definition by power series and as an indefinite integral.

In parallel with the development of the theory some automation tools (tactics) were built with two aims: allowing a significant part of the proofs to be done automatically and enabling the proof assistant to perform the kind of computation that the average person working in this field can do. With these tools, Coq can prove a large number of results involving derivatives and calculate the derivative of functions in a wide class, looking also at the context where this computation is being done. We hope to extend the system in a near future to be able to solve the problem of integrating rational functions, providing both an answer and a proof that this answer is correct.

The basis for this work was chapter 2 of Bishop's book on constructive analysis ([3]). The formalization was built upon the algebraic hierarchy developed at the University of Nijmegen, described in [7] and available in the Coq library, which included most of the results about real numbers that were needed, namely most of sections 1 to 3 of [3] (where real numbers are defined and their main properties are proved); new results about series were formalized, and sections 4 (dealing with continuity, sequences and series of functions), 5 (differential calculus and Taylor's theorem) and 6 (integration and the Fundamental Theorem of Calculus) were completely formalized. Work is in progress regarding section 7 (which is concerned with exponential and trigonometric functions and their inverses).

Our work centered on formalizing the definitions of basic notions in differential and integral calculus, including notions of:

- continuous function;
- derivative;
- differentiable function;
- Riemann integral;
- (convergent) sequence or series of functions;
- Taylor sum and Taylor series of a function.

* This author was supported by the Portuguese Fundação para a Ciência e Tecnologia, under grant SFRH / BD / 4926 / 2001.

Using these definitions, many theorems in this area were formally proved inside Coq; the most important among these were:

- the preservation of continuity through algebraic operations on functions;
- the uniqueness and continuity of the derivative function;
- the derivation rules for algebraic operations on functions and the chain rule for composition;
- Rolle's Theorem and the Mean Law;
- integrability of any continuous function;
- the Fundamental Theorem of Calculus;
- preservation of limits and derivatives through limit operations;
- convergence criteria for series of functions (the ratio test and the comparison test);
- Taylor's theorem.

In section 2 we briefly describe some characteristics of this formalization, including the consequences of working with Coq and of working constructively.

The basic notion which had to be defined and studied at the beginning of the work was the notion of partial function, as most of the common functions of analysis are partial (for example, the logarithm and tangent functions). In section 3 we present the different approaches that were studied and why we chose the one we did.

Section 4 describes how procedures were built that deal with a large class of the most common goals which show up in the area of differential calculus. At the end, we will briefly compare this formalization with similar work already done in other proof systems.

2 Formalizing Mathematics in Coq

Before we go into the specific details of our work, we will briefly discuss some specific Coq issues that influence the way in which our formalization is done.

Coq is a proof tool based on a type system with inductive types called the Calculus of Inductive Constructions (CIC). Through the Curry-Howard isomorphism, proofs are identified with terms and proof-checking with type checking; the construction of a proof then becomes simply the interactive construction of a term which is at the end type-checked.

In the CIC there are two main universes for terms: **Set** and **Prop**. **Set** is meant to be the type for sets and other structures we want to reason about; **Prop** is the type of propositions. There is also an infinite family $\{\mathbf{Type}(i) : i \in \mathbb{N}\}$ such that both **Set** and **Prop** have type $\mathbf{Type}(0)$ and $\mathbf{Type}(i) : \mathbf{Type}(i+1)$, but types in this family will be irrelevant in this paper.

The logic associated with the CIC through the Curry-Howard isomorphism is intuitionistic; this means that to formalize mathematics we must either add the axiom of double negation (in order to be able to work classically) or work constructively. We chose the second alternative, and decided to work following Bishop's approach (see [3]). From our point of view, this is the most general way to work: constructive mathematics results being valid classically, we can always switch to classical reasoning if we want and we will still be able to use all the results we have proved so far[1].

The main characteristic of constructive reasoning is the absence of proofs by contradiction. All proofs have computational content, that is, they provide algorithms to effectively extract witnesses of their statements. So, for example, a proof of an existentially quantified statement $\exists x{:}A.Px$ will amount to an algorithm that presents an element t of type A such that Pt holds.

One of the immediate consequences of this is that some weak form of the Axiom of Choice should be also available for use; that is, if the only way we can prove a statement like $\exists x : A.Px$ is by giving an element satisfying P, then it is also natural to have an operator that allows us to extract such an element from every proof of such a statement.

Unfortunately, Coq does not allow us to define an operator of this kind with type $\mathbf{Prop} \to \mathbf{Set}$ for two different reasons. At a mathematical level, consistency of the system requires such an operator not to be allowed to exist (see [4], pp. 81-83). On the other hand, Coq comes with a program extraction mechanism

[1] An approach following the first alternative was independently chosen by Micaela Mayero, see [11].

(briefly described in Chapter 17 of [4]) which allows programs to be derived from informative proofs; for efficiency reasons, this mechanism assumes that proof objects (living in **Prop**) are irrelevant, as they contain no computational interest. The existence of this operator would undermine this assumption.

Another problem is equality. In type theoretical systems, the natural equality to use is Leibniz equality (given $x, y : A$, $x = y$ iff $\forall P : A \to \mathbf{Prop}.Px \leftrightarrow Py$); however, this turns out to be too strong a concept for most purposes. Therefore, we have to define ourselves a structure with our own equality. This is done through the notion of ***setoid***: a setoid is a pair $\langle S, =_S\rangle$ where $=_S$ is an equivalence relation on S.

For the purpose of formalizing real analysis, equality turns out actually not to be so basic a notion, as it is undecidable on the set of real numbers. However, given two real numbers it is always possible to tell when they are distinct (although if they are not distinct we may never know). This motivates us to use what are called ***setoids with apartness***: setoids where a second relation $\#_S$, called strong apartness, is defined, with the following properties:

- irreflexivity: for all $x : S$, $\neg(x\#_S x)$;
- symmetry: for all $x, y : S$, if $x\#_S y$ then $y\#_S x$;
- co-transitivity: for all $x, y, z : S$, if $x\#_S y$ then either $x\#_S z$ or $z\#_S y$;
- compatibility with equality: for all $x, y : S$, $x =_S y$ iff $\neg(x\#_S y)$.

The last property actually allows us to do away with equality altogether, although it is not usually done.

Functions and relations on setoids are usually required to reflect this apartness relation; that is, if f is a (unary) function from a setoid S_1 to a setoid S_2, then the following property holds: for any two elements $x, y : S_2$,

$$f(x)\#_{S_2} f(y) \to x\#_{S_1} y \ .$$

This property is known as ***strong extensionality*** of f. Predicates in general might not be required to have a similar property (and indeed in many interesting cases they do not), but sometimes the following weaker condition, known as ***well definedness***, is required: for all $x, y : S$,

$$x =_S y \to P(x) \to P(y) \ .$$

From now on, we will use the term "setoid" to mean "setoid with apartness" and denote the equality and apartness relations in a setoid simply by $=$ and $\#$ respectively whenever the carrier set is clear from the context.

At this point we run into another problem of Coq. These definitions work out nicely, but it turns out that if we want to use equality and apartness in a nice way they cannot have type $S \to S \to \mathbf{Prop}$, as would be normal for relations. For this reason, and our desire to use the weak form of the Axiom of Choice which we already mentioned previously, we chose to use also **Set** as the universe for propositions and define our logical connectives to work in this universe with the usual properties.

3 Partial Functions

In our work we only consider partial functions from one setoid to itself. The reason for this is that we are mainly interested in working with real-valued real functions, which satisfy this condition; but generalizing to arbitrary partial functions is quite straightforward and will be done in the near future.

3.1 How to Define Them

Throughout this section A will denote an arbitrary setoid.

The main characteristic of partial functions is that they need not be everywhere defined. Thus, it is natural to associate with each partial function $f : A \not\to A$ a predicate $dom_f : A \to \mathbf{Set}$.

In the algebraic hierarchy which we started from, we have a notion of ***subsetoid*** as being the subset of elements of a setoid S satisfying some property P with the equality relation induced from S; formally, an element of a subsetoid is a pair $\langle x, p\rangle$, where x is an element of S and p is a proof that Px holds. Using this notion, it seems natural to associate every partial function f with a total function on the subsetoid of the elements of A which satisfy dom_f. That is, the type of partial functions will be a dependent record type looking like (in Coq notation):

```
Record PartFunct :=
  {dom    : S->Set;
   dom_wd : (pred_well_def S dom);
   fun    : (CSetoid_fun (Build_SubCSetoid S dom) S)}
```

Here, dom is the domain of the function; the second item of the record simply states that this predicate is well defined[2]; and the third item is a setoid function from the subsetoid of elements satisfying the predicate to S.

Then functional application will be defined as follows: given a partial function f, an element $x:A$ and a proof $H:(dom_f\ x)$, $f(x)$ is represented by the lambda term

$$(\text{fun}\ f\ \langle x, H\rangle)\ ,$$

where fun extracts the subsetoid function from the partial function record.

There are several problems with this definition. One of them is that proofs get mixed with the elements (in the subsetoid construction), which does not seem very natural from a mathematical point of view (where we normally forget about the proof, as long as we know that it exists); another important one is that the terms that we construct quickly get bigger and bigger. For instance, if we have two partial functions $f, g : A \not\to A$ and we want to compose them, the relevant predicate $dom_{g\circ f}$ will look like

$$dom_{g\circ f} := \lambda x{:}A.(\exists H_f{:}(dom_f\ x).(dom_g\ (\text{fun}\ f\ \langle x, H_f\rangle)))\ .$$

Assuming that for some $x : A$ we know that H has type $(dom_{g\circ f}\ x)$, that is, H is a pair consisting of a proof $H_f$of $(dom_f\ x)$ and a proof that $(dom_g\ (\text{fun}\ f\ \langle x, H_f\rangle))$, then, denoting by π_l and π_r the left and right projections, $(g\circ f)(x)$ will reduce to

$$(\text{fun}\ g\ \langle(\text{fun}\ f\ \langle x, (\pi_l\ H)\rangle), (\pi_r\ H)\rangle)\ .$$

This last expression has several unpleasant characteristics, namely it is totally unreadable and very unintuitive; the fact that we are simply applying g to the application of f to x is totally hidden among the pairing and unpairing operations and the proof terms appearing in the expression. Also, if f and g happen not to look at the proof at all (as is the case if they are total functions), they still have to apply projections to recover the argument from the setoid element. This makes the simplification procedure very time consuming.

Thus, a different approach is needed, and we turn to a common alternative which has already been used for example in the Automath system (see for example [2]). As before, we associate to every partial function f the predicate dom_f, but now we identify f with a function of two arguments: a setoid element and the proof that it satisfies dom_f. That is, our type of partial functions will now be:

```
Record PartFunct :=
  {dom      : S->Set;
   dom_wd   : (pred_well_def S dom);
   fun      : (x:S)(dom x)->S;
   fun_strx : (x,y:S)(Hx:(dom x))(Hy:(dom y))
                (((fun x Hx)[#](fun y Hy))->(x[#]y))}.
```

In this definition, dom and dom_wd are as before, but the last item of the record type (which was itself a record) has been unfolded into two components: the function itself (as an element of a product type) and the proof of strong extensionality of that function (which was previously hidden in the type of the setoid function). Given f, x and H as before, functional application now looks like

$$(\text{fun}\ f\ x\ H)\ ,$$

which differs from the previous representation in that we removed one functional application (the pairing operation) and that the element x and the proof H are kept completely separated. This means that, for

[2] Although this is not required from the predicate in order to build the subsetoid, it turned out to be fundamental for our work, namely to prove results about composition the chain rule for derivative, for example.

example, if f is total then it can be computed in a much simpler way, because x is directly available and no extra reduction is needed to get it.

Also comparing with the previous example, the application of a functional composition can be written more nicely given f, g, x and H as

$$(\text{fun } g \ (\text{fun } f \ x \ (\pi_l \ H)) \ (\pi_r \ H)) \ .$$

Notice that in many cases we won't even need to perform any computation on $(\pi_l \ H)$ and $(\pi_r \ H)$, because we won't need to look at the structure of these proofs.

3.2 Working with Function Domains

Once we have partial functions, natural operations with them immediately suggest themselves. The most obvious one (which we have already mentioned) is composition, but algebraic operations (defined point-wise) are also important, at least from the analytical point of view. However, as soon as we try to define this it turns out that it is useful to do some work just with domains.

Since we have identified function domains with predicates, it turns out that what we need is simply a mapping between operations on subsets and operations on logical formulas; that is, given predicates P and Q that characterize subsets X and Y of A we want to define predicates that characterize the sets $X \cap Y$, $X \cup Y$, A, $\emptyset$ and the property $X \subseteq Y$. These can be simply taken to be $\lambda x\!:\!A.(P \ x) \wedge (Q \ x)$, $\lambda x\!:\!A.(P \ x) \vee (Q \ x)$, $\lambda x\!:\!A.\top$, $\lambda x\!:\!A.\bot$ and $\lambda x\!:\!A.(P \ x) \rightarrow (Q \ x)$, respectively. These constructions preserve well definedness (that is, if P and Q are well defined then so will all the predicates defined from them).

As we are concerned with real analysis, it is also important to look at the specific kind of domains we will find. Constructively, it turns out that the most important one is the compact interval, which can be characterized by two real numbers a and b and a proof that $a \leq b$. The predicate corresponding to the interval $[a, b]$ is, of course, simply $\lambda x\!:\!\mathbb{R}.a \leq x \wedge x \leq b$.

The reason for this domain to be so important is that all function properties (continuity, differentiability, etc.) are always constructively defined for compact intervals. Bishop argues (see [3]) that point-wise definitions make no sense in constructive mathematics for the reason that equality is undecidable, and so the information that a function f is continuous at some point x is useless because most times we will not be able to know whether we are exactly at x or not. However, if we work with compact intervals we will often be able to tell that we are inside them (unless we happen to be exactly on the boundary), and so use that information. Another important reason is that constructively it is not necessarily true that e.g. point-wise continuity in a compact interval implies uniform continuity in that interval (a counterexample can be constructed with some extra assumptions, see for example [1]), and so in practice it is more natural to begin with the uniform concept altogether.

The other important kind of domain is the interval. In practice, it is difficult to find examples where we really want to work in a domain which is not an interval or a union of two or three intervals, and the main operations (differentiation, integration) and theorems (Rolle's theorem, Taylor's theorem, the Fundamental Theorem of Calculus) always require that the function(s) involved be defined in an interval.

We model intervals as an inductive type with nine constructors, corresponding to the nine basic kinds of intervals: the real line, the left infinite open or closed, the right infinite open or closed and the finite open or closed on either side. To each kind of interval a constructor is associated: for example, finite, closed intervals are identified with applications of a constructor `clcr`[3] of type $\Pi a, b : \mathbb{R}.\texttt{interval}$. To each of these the obvious predicate is associated, and a property P defined for functions in a compact interval is generalized by

$$P' := (\lambda I\!:\!\text{int}, f : \text{fun})(\forall a, b\!:\!\mathbb{R})((a \leq b) \rightarrow ([a, b] \subseteq I) \rightarrow (P \ [a, b] \ f)) \ .$$

This approach implies that we often have to state very similar lemmas for properties holding in compact intervals and in arbitrary intervals. This is not felt as a disadvantage, however, and is in fact quite close to Bishop's formulation, as most proofs of such properties require distinct reasonings for the compact and the general case.

[3] <u>C</u>losed <u>L</u>eft <u>C</u>losed <u>R</u>ight

4 Automation

We will now discuss what kinds of goals we would reasonably expect to be automatically proved and how successful we have been so far in getting the proof assistant to prove them by itself.

So far we have mainly developed a theory of differentiation, so one of the goals we would expect to pop up very often and which should be automatically proved would be given representations of two functions f and g to prove the relation

$$g \text{ is the derivative of } f. \tag{1}$$

We must also keep in mind that we are doing constructive mathematics, where continuity plays a key role: intuitively, one can argue that all functions that we can define constructively are continuous, but no one reasonably expects this ever to be proved (see [3]); therefore, to make proofs easier, it is typically assumed in the statement of every lemma that all the functions involved are continuous. This means that we expect to come quite often across goals such as

$$f \text{ is continuous.} \tag{2}$$

Finally, the third goal comes as a typical side condition of the lemmas we must apply to prove any statement of the previous two kinds: given a set X and a function f, prove that

$$X \subseteq dom(f). \tag{3}$$

In order to get a better understanding of why goals of type 3 show up so often, we have to look at how we define equality of two functions. This concept is parameterized by domains, that is, for every two functions f and g and subset X of $\mathbb{R}$, we say that f and g coincide on X ($f =_X g$) iff they are both defined in X and they coincide on every point of X, that is, for any element $x : X$ and any appropriate proof terms H_x and H'_x,

$$\forall_{x:X} \forall_{H_x, H'_x}\ f(x, H_x) = g(x, H'_x). \tag{4}$$

Two comments are due on this definition:

- The inclusion of X in the domains of both f and g is essential if we want to get something that looks like an equality, namely a transitive relation. If we did not require this condition then every function would be equal in every set to the undefined function, and no substitution properties[4] would hold.
- The reason why we explicitly state that f and g are defined in X is to make proof development easier. This way, we are left with three independent goals to prove: the two inclusions and (4), which we can prove independently.
  If we did not state the inclusion explicitly, then we would only have to prove

  $$\forall_{x:X} \exists_{H_x, H'_x}\ f(x, H_x) = g(x, H'_x) \ ,$$

  which differs from the third one in that the proof terms are existentially quantified. However, the existential quantifiers make this goal much more difficult to prove and less suited to automation, which is why we chose the first approach.

We begin by considering goals like (3). Typically, they are proved by looking at the algebraic structure of f and the knowledge that inclusion is preserved through algebraic operations, that is, if $X \subseteq dom(f_1)$ and $X \subseteq dom(f_2)$ then $X \subseteq dom(f_1 + f_2)$ and similarly for other operations. There are some side conditions that have to be verified when division shows up, but these too are usually taken care of by one of a small group of lemmas.

When f has no algebraic structure, there is also a small number of results we can use, namely the facts that constant and identity functions are defined everywhere and that any function which is continuous in X, has a derivative in X or is the derivative in X of some function is also defined in X.

With this knowledge, we can (and we have) easily implement a tactic with the Coq `Hints` mechanism which simply looks at the form of the goal and chooses the right lemma to apply from a not too big list. This turns out to be satisfactory enough for small to medium sized goals, although it doesn't always work

[4] Like if $f =_X g$ and f is continuous in X then so is g.

when the structure of f is too complicated. In those situations, typically the user has to break down f in smaller parts by himself, and then invoke the automatic prover.

Goals like (2) work in quite a similar way, and have been treated in the same way.

When we turn to goals like (1), however, things turn out to be quite different. From a naive perspective, we would expect this situation to be similar to the previous ones, as we intuitively reason in this situation by cases using a small set of lemmas—the derivation rules. However, when we analyze the situation more carefully it is not as simple as it looks, as we show with a small example. Let f and g be functions everywhere defined by the rules $f(x) = 3x + 4$ and $g(x) = 3$, respectively. If we want to prove $f' = g$, then we would like to begin by applying the derivation rule for the sum; however, in order to do this we also need to have a function that is the sum of two other functions on the right side, and this is not the case. Hence we are stuck.

The trick to to this is, obviously, to replace g by what we get if we differentiate f by using the differentiation rules in this case, by h such that $h(x) = 3 * 1 + 0$. Then we can easily prove that $f' = h$ and we are left with the goal $g = h$, which is also easy to prove. The problem, then, amounts to finding h.

Intuitively, we would like to make some kind of recursive definition that looks at the algebraic structure of f. However, there is no inductive structure in the class of partial functions, so this is not directly possible. However, the Coq tactic language allows us to do something similar: we meta-define (that is, we define in the meta-language) an operator that looks at the structure of f and correspondingly builds h. This operator recognizes algebraic operations, functional composition and can look at the context for relevant information (for instance, if there is a hypothesis stating that for some functions f_1 and f_2 the relation $f_1' = f_2$ then it will use f_2 as a derivative for f_1); however, the proof is still left to be done by the user.

Another, and more powerful, approach is to use reflection (a method which is described in full detail in [8]). We select among the class of all partial functions those whose derivative we know how to compute, and model this as an inductive type $\mathcal{PF}$. This type will have not only constructors for constant and identity functions and algebraic operations on these, but also two special constructors that allow us to add any function about which we know something from the context. This will allow us, for instance, to prove that $(2f)' = 2g$ if we know from the context that $f' = g$.

On $\mathcal{PF}$ we will define two operations: a canonical translation map $[\![\cdot]\!] : \mathcal{PF} \to (\mathbb{R} \not\to \mathbb{R})$ to the real-valued partial functions and a symbolic differentiation operator $' : \mathcal{PF} \to \mathcal{PF}$ with the property (stated as a lemma) that for every $s : \mathcal{PF}$

$$[\![s']\!] \text{ is the derivative of } [\![s]\!]. \tag{5}$$

Our problem now amounts to the following: given a function f, how do we determine an element $s : \mathcal{PF}$ such that $[\![s]\!] = f$? That is, how can we define a (partial) inverse to $[\![\cdot]\!]$? Again, this is done at the tactic level in Coq: we meta-define an operator by case analysis that looks at the structure of f and breaks it down; whenever it finds an algebraic operator, constant or identity function, it replaces this by the corresponding constructor in $\mathcal{PF}$; whenever it finds a function that it knows nothing about (that is, an expression like "f") it tries to find an hypothesis in the context that allows it to use one of the two special constructors. If everything goes well, we get indeed an element s with the required property; otherwise we get an error message.

With these tools we can then write down our tactic as follows: given f and g,

1. Find $s : \mathcal{PF}$ such that $[\![s]\!] = f$;
2. Compute s';
3. Replace f by $[\![s]\!]$;
4. Replace g by $[\![s']\!]$;
5. Prove that $[\![s]\!] = f$;
6. Prove that $[\![s']\!] = g$;
7. Apply lemma (5) to prove that $[\![s']\!]$ is the derivative of $[\![s]\!]$.

Steps 3 and 5 may seem superfluous, as s was constructed so that $[\![s]\!] = f$ would hold. The problem, however, is that we did not define this construction as an element of type $(\mathbb{R} \not\to \mathbb{R}) \to \mathcal{PF}$ (because no such element with the required properties exists), so we cannot prove anything in general about this operation. Still, step 5 turns out to be trivial, as simplification on $[\![s]\!]$ yields f and we just have to invoke reflexivity of equality.

Step 6 is the tricky one. In the most cases, this will reduce to proving some inclusions of domains (which we have already automated) and then equality of two algebraic expressions (which the `Rational` tactic, described in [8], can normally deal with); in some cases, however, this step leaves some work to be done, for example if the equality between $[\![s']\!]$ and g relies on the fact that any two derivatives of a given function must coincide. Even in this cases, however, experience has shown that the goal has been much simplified, so that we do profit from this tactic.

At the present moment, the biggest limitation of this tactic is that it cannot deal with division or functional composition. However, experience shows it to be much more efficient (both regarding computation time and the size of the constructed proof-terms) than the first approach. Also the limitations turn out not such a big problem as they could seem, actually, because we can always add the relevant steps as hypothesis to the context and prove them later; but they still are limitations, and it is interesting to see why we can't deal with these cases in the same way as we dealt with the others.

When we look at the constructive rule for derivation of a division or composition of two functions, they turn out to differ from the other rules in that they have some side conditions that have to be met: as an example, to apply the rule for division, we have to prove that the absolute value of the denominator of the fraction we want to derive is always greater than some positive constant. In order to prove that this side conditions always hold (which we have to do if we want to prove something like $\forall_s [\![s']\!] = [\![s]\!]'$), we have to add in the constructor of $\mathcal{PF}$ corresponding to division an argument stating something about the interpretation of one of the other arguments. But this is not possible in Coq, because we cannot simultaneously define an inductive type and a recursive function over that type (although type theory allows us to do this, namely in this situation).

The case of composition is even worse, as one of the goals we get says something about one function being the derivative of another in an unknown interval. One way to solve this problem would be to make our tactic interactive in some way, but there is no obvious way to do this.

Presently, as we said, these limitations turn out not to be such a big deal. Division is not such an important operation when we work constructively, as most situations that use division can be rewritten so as to use only multiplication; and for composition we can usually achieve our goals by adding hypothesis to the context and applying the chain rule by hand. When none of this works, we can still rewrite the function on the right-hand side of the goal with the first operator we define and proceed by hand.

5 Related Work

This same fragment of real analysis has already been formalized in some systems by different people. We will now briefly describe these formalizations and how they differ from ours.

Micaela Mayero (see [11]) has formalized differential calculus in Coq, including notions of (point-wise) continuity and differentiability, derivation rules, and some work on transcendental functions. However, she does not treat integral calculus or more general theorems like Rolle's theorem. This is because her motivation is not formalizing real analysis in itself, but showing how such a formalization can be used for other purposes, whence she develops just the theory that she needs for her examples. For the same reason, she argues that it makes more sense for her to work classically—which makes her work totally distinct from ours.

Mayero's treatment of partial functions also differs from ours. As we do, she always associates a domain with every function; however, they are kept completely separated: functions have type $\mathbb{R} \to \mathbb{R}$, domains $\mathbb{R} \to$ **Prop**, and the domain is always explicitly stated in the formulation of the lemmas. Although this makes it possible to write things down in a way closer to usual mathematical notation (that is, $f(x)$ instead of $f(x, H)$ or something similar) it does have the disadvantage that you can write down things like $\frac{1}{0}$, although it is not clear what they mean.

In the PVS system, Bruno Dutertre (in [5]) has also developed a classical theory of real analysis, including the main definitions in differential calculus. Building upon this work, Hanne Gottliebsen built a library of transcendental functions described in [9], where she defines exponential, logarithmic and trigonometric functions, proving similar results to ours. She then defines an automatic procedure to prove continuity of a large class of functions, which works in a similar way to ours, and shows how it can be used interactively with Computer Algebra systems to guarantee the correctness of applications of the Fundamental Theorem of Calculus.

John Harrison's HOL-light system (described in [10]) is another proof assistant that comes with a library of real analysis: once again, the reasoning in this system is classical. The results included in this library include the usual results on preservation of continuity through algebraic operations, derivation rules, Rolle's theorem and the Mean Law.

Also included in the system is a library of transcendental functions, where exponential and trigonometric functions are defined as power series and their inverses as inverse functions. Finally, integration is defined and the Fundamental Theorem of Calculus is proved.

6 Conclusions and Future Work

We have shown how to formalize an important fragment of constructive real analysis and how to use this formalization to build automation tools that can (partly) solve some problems in this area, by providing not only an answer but also a proof that this answer is correct. Presently we deal only with differentiation in a restricted class of functions, but work is being done to generalize the setting to all the elementary transcendental functions. We hope to complete this work with similar results regarding integration, namely by providing a way to integrate rational functions and prove the result correct.

In doing so, we have also shown that it is possibly to build and use modular libraries of mathematical formalizations, as our work was done using a library of real numbers which was already developed and to which no changes were made (although some results had to be added dealing with specific problems). We have also provided evidence to Bishop's claim that it is indeed possible to do useful mathematics without the double negation rule.

Acknowledgments

Support for this work was provided by the Portuguese Fundação para a Ciência e Tecnologia, under grant SFRH / BD / 4926 / 2001.

The author would also like to thank H. Barendregt, H. Geuvers and F. Wiedijk both for the many discussions throughout the development of this work, which contributed to its successful outcome, and for their suggestions regarding the contents and form of this paper.

References

1. Beeson, M., *Foundations of constructive mathematics*, Springer-Verlag, 1985
2. Benthem Jutting, L. S. van, *Checking Landau's "Grundlagen" in the Automath System*, in Nederpelt, R. P., Geuvers, J. H. and de Vrijer, R. C. (Eds.), *Selected Papers on Automath*, North-Holland, 1994
3. Bishop, E., *Foundations of Constructive Analysis*, McGraw-Hill Book Company, 1967
4. The Coq Development Team, *The Coq Proof Assistant Reference Manual Version 7.2*, INRIA-Rocquencourt, December 2001
5. Dutertre, B., *Elements of Mathematical Analysis in PVS*, Proc TPHOLS9, LNCS 1125, Springer, 1996
6. Geuvers, H. and Niqui, M., *Constructive Reals in Coq: Axioms and Categoricity*, in Callaghan, P., Luo, Z., McKinna, J. and Pollack, R. (Eds.), *Proceedings of TYPES 2000 Workshop*, Durham, UK, LNCS 2277
7. Geuvers, H., Pollack, R., Wiedijk, F. and Zwanenburg, J., *The algebraic hierarchy of the FTA project*, in *Calculemus 2001 Proceedings*, Siena, Italy, 13-27, 2001
8. Geuvers, H., Wiedijk, F. and Zwanenburg, J., *Equational Reasoning via Partial Reflection*, in *Theorem Proving in Higher Order Logics*, 13th International Conference, TPHOLs 2000, Springer LNCS 1869, 162-178, 2000
9. Gottliebsen, H., *Transcendental Functions and Continuity Checking in PVS*, in *Theorem Proving in Higher Order Logics*, 13th International Conference, TPHOLs 2000, Springer 2000
10. Harrison, J., *Theorem Proving with the Real Numbers*, Springer-Verlag, 1998
11. Mayero, M., *Formalisation et automatisation de preuves en analyses relle et numrique*, PhD thesis, Universit Paris VI, dcembre 2001

The DOVE Approach to the Design of Complex Dynamic Processes

Brendan Mahony

Defence Science and Technology Organisation,
Department of Defence,
PO Box 1500, Edinburgh, South Australia 5111,
Australia

Abstract. DOVE is a Isabelle-based, graphical tool for the trusted design and analysis of state machine designs. Work is currently underway aimed at increasing the scope of DOVE to include complex, dynamic and real-time processes. This paper describes a refinement approach to the design and analysis of complex processes, including a formal development of the approach in Isabelle. It is argued that the addition of a network feedback operator and associated refinement rules makes the refinement calculus of Back a powerful development tool for process networks, allowing essentially arbitrary decomposition of network properties over subcomponents. The feedback operator is shown to be 'compositional' with respect to an implementation language of digital input/output processes with local state. Some simple examples are considered.

1 Introduction

The effective and economical development of complex, dynamic and real-time systems is a matter of considerable interest to the Australian Defence engineering community and to the engineering community in general. This is especially so where the systems under development serve critical functions and require the highest levels of trust and assurance when fielded into service. Many existing regulatory authorities demand the highest levels of mathematical rigor in the analysis of security-critical and safety-critical systems, levels of rigor which present a considerable challenge to the current state-of-the-art in the mathematical analysis of complex systems. The DOVE design and analysis environment aims to offer developers a powerful and usable tool for supporting the development and presentation of assurance cases for critical systems.

The current version of the DOVE tool [7] comprises a sophisticated state-machine reasoning tool developed in the Isabelle/HOL environment [25] and a graphical user environment which supports and encourages a highly visual approach to the design and analysis of state-machine based systems. Although the use of a general theorem proving approach to treating the properties of state-machines allows the treatment of larger state-spaces than the competing technology of model-checking, the state-machine formalism still does not scale well to large systems with complex interacting components. Nor does it allow, or even offer the prospect of, the treatment of real-time or physical process components. The effective treament of such components requires the adoption of an analog process model, or at least the inclusion of analog aspects in a discrete model such as adopted in so-called *hybrid*-systems approaches.

A common approach to the problem of composing interacting systems is the dataflow network. It enjoys widespread recognition and has been adopted in many existing informal development approaches. Dataflow networks also admit a natural and highly intuitive visual representation as directed graphs in which the nodes represent system components and the edges represent informational or even physical flows. Finally, dataflow networks readily allow the treatment of analog components, it is only necessary to make use of continuous functions to model some or all of the dataflows involved.

The work presented in this paper considers the use of dataflow networks to provide a formal basis for the design and tractable analysis of networks of interacting processes, both digital and analog.

1.1 Compositional development

The separation of formal specifications into assumptions about the environment of a process and commitments made by the process has a venerable formal methods pedigree [11], stretching back through Hoare's axioms [10] to the germination of the idea in work by Floyd [9] and even Turing [29]. By supporting the meaningful specification of open systems, the assumption/commitment approach has become the cornerstone of many compositional approaches to treating complex systems.

The adoption of what we term the *assumption/commitment paradigm* in the development of process networks may be traced to Misra and Chandy [20], who used assumption/commitment specifications on system traces to develop a compositional method for the verification of safety properties in networks of CSP-like processes. Subsequently, many other authors have used similar approaches to develop methods for the verification of various subclasses of process properties. Some such approaches are due to Pandya and Joseph [26]; Abadi and Lamport [1]; Stølen, Dederichs, and Weber [27]; and Stølen [28]. In every case however, the methods offer complex verification conditions and (with the exception of that of Stølen) allow the treatment of at best a restricted class of process properties. Furthermore, they rely for their effectiveness on specialised (and somewhat baroque) process models with narrow areas of application. The primary reason for such complexity and restrictions lies in the difficulty of defining a general, compositional model of network construction.

One difficulty in modeling network construction has been the common approach of defining a parallel hookup operator which includes both parallelism and feedback capabilities. The complexity of such all-purpose operators tends to overwhelm the search for tractable approaches to reasoning. As observed by Katis *et al* [12], it is preferable (at least in the abstract) to define separate parallel and feedback hookup operators. Another difficulty has come from the tendency to treat feedback in terms of recursive function theory. A more promising approach has been suggested by Katis *et al* [12]. They describe a relational feedback operator based directly on a naive notion of fixed points. As demonstrated in the remainder of this paper, making use of a separate feedback operator based directly on this naive notion of fixed points greatly improves the tractability of reasoning about networks of processes.

1.2 Refinement

A separate development of the assumption/commitment paradigm has seen the utilisation of predicate transformer semantics in support of compositional development methods for sequential programs. *Weakest precondition* program semantics were first suggested by Dijkstra [8] and have been blended successfully with the assumption/commitment paradigm independently by Back [4], Morgan [21], Morris [23], and Nelson [24]. These formalisms have much in common and are referred to collectively as the *refinement calculus*.

The refinement calculus is a broad-spectrum, specification/programming language together with a collection of *refinement rules* that support top-down design. High level specifications are *refined* to mixtures of specification and program code and finally into pure program code. The pure code subset of the refinement calculus is called the *implementation language* and will vary with the problem to which the refinement calculus is being applied. For example, Morgan's program refinement calculus adopts Dijkstra's guarded commands as its implementation language.

The refinement calculus approach has been used successfully in several case studies in the specification and design of real-time and reactive processes [13, 15, 17, 18]. The purpose of this paper is to formally define an extension of Back's predicate-transformer model so as to allow its use in the treatment of interacting processes. By formally, in this case, we mean that the extension has been developed using a formal modelling tool, namely Isabelle's HOL modelling environment [25].

Following Katis *et al* we describe a network construction model which allows processes to be hooked up in sequence, parallel, and via feedback. The sequence operator is well known, being originally described by Dijkstra. The parallel operator has been the subject of considerable interest in recent years, first defined by Martin [19] and then investigated in detail by Back and Butler [5, 6] and also Mahony [14]. The feedback operator is partially a contribution of this paper, having been suggested by the relational operator of Katis *et al.* Following the usual refinement calculus approach we define a collection of novel refinement laws involving these operators that support the top-down development of process networks from abstract specifications to concrete implementations.

The resulting refinement environment represents a powerful tool for the analysis of both liveness and safety properties of dynamic proceses. Furthermore, it is a tool which does not depend for its effectiveness on a particular model of computation. In particular, it is in principle possible to adopt either digital or analog

Fig. 1. Example process network.

process models and even to mix them. This represents a clear advance in analytical completeness over the restrictive classes of properties and systems treated by most of the methods described above and a clear advance in tractability over the more complete, though still model-specific, method of Stolen [28].

1.3 Summary of paper

The balance of the paper has the following outline. In Section 2 the basics of predicate transformer algebra are introduced. In Section 3 the three network hookup operators are defined and refinement rules introduced. In Section 4 an implementation language for networks of *IFO machines* is introduced. This language is used in Section 5 to present some examples in the use of the refinement calculus. Finally, the results of the paper are summarised in Section 6 and the network refinement calculus compared to existing network verification methods.

2 Predicate transformer basics

This section briefly introduces the basics of predicate transformer algebra, as presented by Back and von Wright [3]. The formal text in this paper follows the syntax and conventions of the Isabelle/Isar implementation of HOL [25]. In particular, proofs are presented in the Isar style [30] of proof-programming. Briefly, the proof justifications fall into three broad categories. The keywords *rule*, *intro*, and *elim* indicate the use of theorems as inference rules. The keyword *simp* indicates the unwinding of definitions. The keywords *auto*, *fast* and *blast* indicate the use of automated proof procedures. In general, the full Isar proof script is presented, but where the full proof is particularly tedious we elide it, offering instead a brief informal justification.

This paper aims to address the high-assurance design and analysis of complex processes. Processes are viewed as hierarchical networks of process elements communicating along dedicated channels. Such processes may be represented using annotated graphs such as that depicted in Fig. 1. Various forms of polygons are used to represent classes of network elements and directed arcs are used to represent information flows or process components. This ability to render process networks in a graphic form is an important tool for communicating their component structure and will form the basis of a graphical user interface for interacting with process hierarchies in a forthcomming version of the DOVE tool. We make judicious use of it throughout this paper.

The properties of processes are expressed through predicates. The reader is assumed to be familar with the algebraic properties of predicates, but briefly a *predicate* $\phi::\mathbb{B}^{\Sigma}$ is a boolean-valued function of process state Σ. The usual boolean operators are lifted to act on predicates, with the following operator precedents: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$. The standard boolean order is lifted pointwise to define the *entailment* order ($\Rrightarrow$) on predicates.

We identify three basic models for describing processes/specifications that, given inputs from Σ, construct outputs from Γ.

The simplest model of process is as a logical function $f::\Sigma \rightarrow \Gamma$ from input states Σ to output states Γ. Functions allow us to describe from each input precisely the desired output. Such detail is of course necessary for an implementation, but is often tiresome in the early stages of design.

An easier option is to specify a range of allowed results. This can be done using a *relation* $R{::}\Sigma \leftrightarrow \Gamma$, which is a function from input states Σ and output states Γ to the booleans. The natural lifting of the boolean order to relations yields the entailment order on relations (also written $\Rrightarrow$) which increases as more choices are added to a relation. The range of choices allowed by a relation is known as the *nondeterminism* exhibited by a relation. A relation $S{::}\Sigma \leftrightarrow \Gamma$, such that $S \Rrightarrow R$, may be thought of as an implementation of R.

In the abstract, relational entailment offers a simple, and therefore attractive, model for treating the design process, but in practice relational verifications tend to be hard to deal with and to involve numerous repetitive and complex calculations. In addition, it can be difficult to treat incompleteness and inconsistency in specifications in an entirely satisfactory manner. These problems can be overcome by adopting the *predicate transformer* as the basic process model. Predicate transformers were introduced by Dijkstra [8] as a generalisation of relations.

A *predicate transformer* $p{::}\Sigma \mapsto \Gamma$ is a function from predicates over output states $\mathbb{B}^{\Gamma}$, which we refer to as *effects*, to predicates over input states $\mathbb{B}^{\Sigma}$, which we refer to as *presumptions*.

2.1 The refinement calculus

Predicate transformers offer a richer algebraic structure in which to model and analyse computational mechanisms, than do either functions or relations. Indeed Dijkstra seems to have found predicate transformers *too* rich in structure and immediately began suggesting "healthiness criteria" intended to restrict attention to those predicate transformers sufficiently relation-like in nature to be comprehended using his existing relational intuitions. As our understanding of the algebra of predicate transformers has grown, we have gradually come to appreciate the power of such unintuitive features as magic, coercion, and angelic non-determinism, however one healthiness criteria remains. We make use only of those predicate transformers which are monotonic with respect to entailment, since these are rational in the sense that stronger presumptions are required to achieve stronger effects.

In the following, we use the term *process* as a synonym for monotonic predicate transformer, since this makes it easier to convey the intuitions behind the predicate transformer model.

defs *monotonic_def*: $monotonic\ p \mathrel{\widehat{=}} \forall \phi\ \psi \bullet \phi \Rrightarrow \psi \Rightarrow p\ \phi \Rrightarrow p\ \psi$

Though we do not present the proofs, all of the operators presented in this paper construct monotonic predicate transformers.

The pointwise lifting of the entailment ordering is called *refinement* ($\_ \sqsubseteq \_$) and is read "is refined by".

defs *refbyeq_def*: $p \sqsubseteq q \mathrel{\widehat{=}} \forall \phi \bullet p\ \phi \Rrightarrow q\ \phi$

The term refinement alludes to a view of top-down design as the process of removing the "impurities" of incompleteness and nondeterminism in a specification until all that is left is the "pure" code which was originally intended. This view is supported perfectly by the refinement relation since every refinement of a process is able to achieve all of its effects under the same or weaker presumptions. Thus from a process design standpoint, it is always acceptable to replace a process with a more refined process.

Much of process design can be viewed in terms of finding solutions to problems in process refinement and the algebra of predicate transformers provides an ideal tool (we call it the *refinement calculus*) for actually calculating solutions to design problems. This calculational facility often allows predicate transformer based verification systems to be simpler than corresponding relational systems. In the refinement calculus, verification laws tend to require fewer human-supplied parameters (many parameters can be replaced by calculated most-general solutions) and fewer verification conditions (most-general solutions are solutions by construction). In fact, these laws tend to be so much simpler that we call them *refinement* laws, so as to focus attention on their use as design development tools rather than design checking tools.

The approach to network design taken in this paper involves the definition of a collection of predicate-transformer operators that allow the modeling of process designs and implementations, together with refinement laws for introducing and eliminating these operators during design development.

Again, though the proofs are not presented, all of the operators presented in this paper are monotonic with respect to the refinement relation. This property is called *vertical compositionality* by Zwiers *et al* [31].

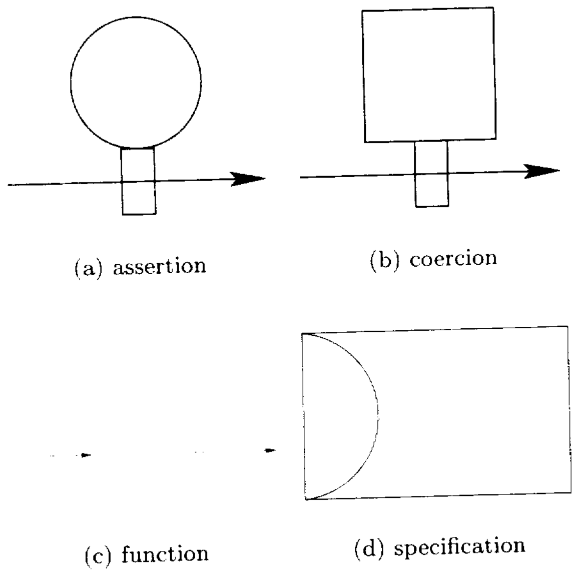


Fig. 2. Predicate transformer embeddings.

Compositionality is critical to the utility of the entire refinement approach, since it allows network designs to be decomposed in such a way that the subcomponents may be developed in isolation, possibly by separate design teams.

A second aspect of compositionality noted by Zwiers *et al* is *horizontal* compositionality. This requires an operator to preserve implementations in the sense that when its arguments are implementations its result is also an implementation. We call such operators *code-preserving*. It is important to realise that, in the context of the refinement calculus, horizontal compositionality is a function of the implementation language. All refinement calculus operators must be monotonic, but only program operators need be code-preserving.

Following Morgan [22] we express refinement laws in a vertical style which emphasises the transformation aspects of the refinement process. In the vertical style, the specification is presented above a design, separated by a horizontal rule and with any side-conditions to application of the law placed to the right of the rule. For example, we write

$$\frac{p}{q} \quad G$$

for the proposition $G \Rightarrow p \sqsubseteq q$.

2.2 Predicate transformer embeddings

Predicates may be embedded as predicate transformers in two ways, either as an *assertion* about the process state or as a *coercion* of the process state. Assertions and coercions can be represented as lollipop-style nodes on a network graph, as depicted in Fig. 2(a) and 2(b).

defs
assert_def: $\{A\} \mathrel{\hat=} \lambda\phi \bullet A \wedge \phi$
coerce_def: $[A] \mathrel{\hat=} \lambda\phi \bullet A \Rightarrow \phi$

Assertions are refined by weakening their predicate and coercions by strengthening it.

lemma *assertW*: $\dfrac{\{\phi\}}{\{\psi\}} \quad \phi \Rrightarrow \psi$
by (*simp add*: *pand_def*)

lemma *coerceS*: $\dfrac{[\psi]}{[\phi]} \quad \phi \Rrightarrow \psi$
by (*simp add*: *pimp_def*)

Simple functions become point-replacement operators when embedded as predicate transformers. They are represented on network graphs as diamond shaped nodes, see Fig. 2(c).

defs *function_def*: $\langle f \rangle \mathrel{\hat{=}} \lambda \phi\ a \bullet \phi\ (f\ a)$

Relations may be embedded as predicate transformers to give abstract specifications of desired relations between inputs and outputs. Following Morgan [21], we introduce the specification statement as the primary method of expressing relational specifications. In addition to a relation between inputs and outputs (the *commitment*), the specification statement also includes an *assumption* about the properties of inputs. Specification statements are represented graphically by rectangular nodes, partitioned into assumption and commitment compartments, as shown in Fig. 2(d).

defs *spec_def*: $[\,A \,/\, E\,] \mathrel{\hat{=}} \lambda \phi\ a \bullet A\ a \wedge (\forall\, b \bullet E\ a\ b \Rightarrow \phi\ b)$

The main refinement laws dealing with specification statements allow the utilisation of the assumption when strengthening the commitment and the weakening of the assumption.

lemma *commitS*: $\dfrac{[\,A \,/\, E_1\,]}{[\,A \,/\, E_2\,]} \quad (\lambda a\ b \bullet A\ a \wedge E_2\ a\ b) \Rrightarrow E_1$
by (*auto*)

lemma *assumeW*: $\dfrac{[\,A_1 \,/\, E\,]}{[\,A_2 \,/\, E\,]} \quad A_1 \Rrightarrow A_2$
by (*simp*)

A deterministic specification is refined by the corresponding function.

lemma *funI*: $\dfrac{[\,A \,/\, \lambda a\ b \bullet b = f\ a\,]}{\langle f \rangle}$
by (*simp*)

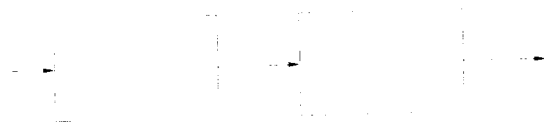
3 Network constructors

We support three methods for hooking up the inputs and outputs of processes, as shown in Fig. 3. The first method is sequential hookup, which is modeled by function composition. The second is parallel hookup, which is modeled by the predicate transformer product operator. The third is feedback hookup, which is essentially coerces one of a process's outputs to have the same value as one of its inputs.

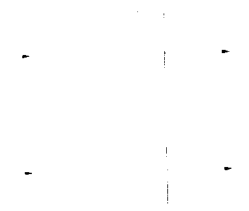
3.1 Sequential hookup

Composing processes sequentially is a simple matter of passing the outputs of one to the inputs of the second. In the functional model this is achieved through function composition, in the relational through relational join, and in the predicate transformer through reverse function composition. Sequential hookup is monotonic, associative, and its identity hookup is the identity function **1**.

defs
fseq_def: $f \gg g \mathrel{\hat{=}} \lambda a \bullet g\ (f\ a)$
rseq_def: $R \gg S \mathrel{\hat{=}} \lambda a\ b \bullet (\exists\, c \bullet R\ a\ c \wedge S\ c\ b)$
seq_def: $p \gg q \mathrel{\hat{=}} \lambda \phi \bullet p\ (q\ \phi)$



(a) sequence



(b) parallel (c) feedback

Fig. 3. Hookup mechanisms.

The technique for sequential decomposition of a specification, is to first express the commitment as a sequential composition.

lemma *midI*: $$\frac{[A / E \gg F]}{[A / E] \gg [\lambda b \bullet \exists a \bullet A\ a \wedge E\ a\ b / F]}$$
by (*auto*)

We also require rules for introducing coercions and for transferring information from coercions to specifications.

lemma *coerceI*: $$\frac{p}{[\phi] \gg p}$$
by (*simp add*: *pimp_def*)

lemma *assumeS*: $$\frac{[\phi] \gg [A / E]}{[\phi] \gg [\phi \wedge A / E]}$$
by (*simp add*: *pimp_def pand_def*)

3.2 Parallel hookup

Hooking up two processes in parallel creates a single process which accepts pairs of inputs and produces pairs of outputs. This is straightforwardly modeled in the function model of computation by the function product operator.

defs *fprod_def*: $f_1 \times f_2 \mathrel{\hat{=}} \lambda(a_1, a_2) \bullet (f_1\ a_1, f_2\ a_2)$

It is equally straightforward to define product operators for predicates and relations. However, lifting the notion to predicate transformers proved more difficult and a number of approaches [14] were proposed before agreement was found. The basic idea of this predicate-transformer product is quite straightforward, that is to map effect predicates over a product space to presumption predicates by mapping the individual components separately. Thus, the process $p_1 \times p_2$ maps $\psi_1 \times \psi_2$ to $(p_1\ \psi_1) \times (p_2\ \psi_2)$. Unfortunately, few predicates over a product space are a product of predicates over the individual components. However this basic premise can be used to define the predicate-transformer product using a technique similar to the "sum-of-the-rectangles" approach to defining area in geometry. The image of ϕ under $p_1 \times p_2$ is the union of the images of all the "rectangles" contained in ϕ.

defs *product_def*: $p_1 \times p_2 \mathrel{\hat{=}} \lambda\phi \bullet \bigvee\ \psi_1\ \psi_2 | \psi_1 \times \psi_2 \Rrightarrow \phi \bullet (p_1\ \psi_1) \times (p_2\ \psi_2)$

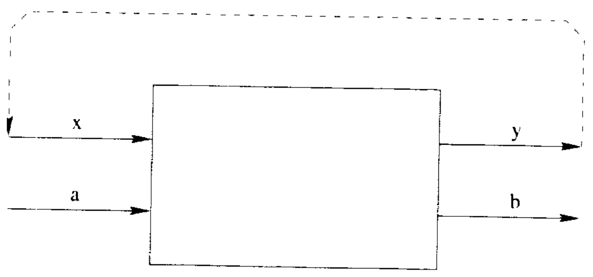


Fig. 4. Introducing a feedback loop.

This definition was first proposed by Martin [19] and has been analysed in some detail by Back and Butler [5, 6]. Another possible approach to defining a product is to make use of a relational decomposition property [19] to lift the relational product operator [14]. It turns out that both approaches yield the same operator.

The basic technique for introducing a parallel hookup is to decompose the assumptions and the commitments according to the desired subcomponents.

lemma *spec_prod*: $\dfrac{[\, A \times B \,/\, E \times F \,]}{([\, A \,/\, E \,]) \times ([\, B \,/\, F \,])}$
by (*auto simp add*: *pprod_def*)

Since, as already conceded, few predicates (or relations) are of rectangular form, this is a highly restrictive approach to introducing products. However there are some points that can be made in favour of this situation.

Primarily, it must be noted that there should be no great imperative to decompose processes in parallel at an early stage of design. In fact, in the general case, this is likely to be a highly ambitious aim. Consider what such a decomposition implies about a design, namely that the subcomponents of a given process admit such a strong decoupling of their behaviour that their further development may be done in complete isolation. Seen in this light, it is clear that the introduction of products should not be forced, but rather that products should be allowed to arise naturally from the design process as the elements of the design become more concrete and determined. Indeed, some process components may exhibit such strong coupling of behaviour that it never becomes convenient to explicitly separate them.

An artificial imperative to perform such decompositions has been introduced into a number of existing approaches due to the coupling of the product and the feedback operators. In order to make use of the properties of a feedback loop in a design it is thus necessary to "discover" a suitable decomposition of the process components. This forms a major barrier to the use of such methods and is a strong argument in favour of a decoupled approach to modeling parallel processes.

3.3 Feedback hookup

The third method of hooking up the inputs and outputs of processes is through the introduction of feedback. The essence of feedback, as it is for iteration and recursion, is the construction of a fixed point. To see this, consider the simple (abstract) network element depicted in Fig. 4. The effect of introducing a feedback loop (depicted as a dashed line) from the output y to the input x is to equate their two values, necessitating the discovery of a fixed point of the process when viewed as an input/output transformer. The *only* difference is that in the cases of iteration and recursion the desired fixed point is a program while in the feedback case the desired fixed point is some user-defined data structure. This of course has profound implications for modeling feedback since it cannot be treated straightforwardly through the existing, complex models for treating iteration and recursion. Perhaps the very focus on highly-developed fixed-point theories for treating iteration and recursion has been something of a distraction in the treatment of feedback (we discuss other approaches in Sect. 6). In fact, as was pointed out clearly by Katis *et al* [12], the situation with feedback is actually much simpler. The introduction of a feedback loop may be viewed quite simply as the strengthening of an existing specification to require that an output have the same value as a given input. It is this model

Fig. 5. Definition of feedback.

of feedback which we adopt, embodying it in the predicate transformer model by (as depicted in Fig. 5) introducing the appropriate coercion and hiding the feedback variable so as to protect it from outside effects.

defs *feedback_def*:
$[p] \phi\ a \mathrel{\hat=} (\forall x \bullet (p \gg [(\lambda\ x' \bullet x' = x) \times \text{true}\ ])\ (\text{true} \times \phi)\ (x, a))$

Since a feedback loop introduces a hidden coercion, it is important to have a clear intuition as to the potential effects this may have. The first and most obvious danger is that the component process may have no fixed points, making the feedback process inconsistent and therefore unimplementable. A more subtle danger is that the component process may have many fixed points, even if it is itself deterministic. Thus a feedback process may be nondeterministic even if its component process is deterministic.

Introducing a feedback loop is simply a matter of expanding the input/output spaces of a specification to accommodate the feedback component.

lemma *spec_fbI*: $\dfrac{[A\,/\,E]}{[\,[\lambda(x,\, a) \bullet A\ a\,/\,\lambda(x,\, b)\ (x',\, b') \bullet E\ b\ b'\,]\,]}$
by (*simp*)

After applying *spec_fbI*, the designer is free to use other refinement laws to introduce the desired properties of the feedback component. The important question in this is how the designer should go about introducing assumptions about the behaviour of the feedback component. Our suggested approach harks back to Morgan's [21] original arguments in favour of the positive applications of miracles. We propose that the designer introduce the desired properties as *coercions* of the feedback component on the *input* side so as to allow them to be used as assumptions in subsequent development. Such refinements would conform to the following general outline.

have
$[A\,/\,E]$
$\sqsubseteq$
$[\,[\lambda(x{::}\alpha,\, a) \bullet A\ a\,/\,\lambda(x{::}\alpha,\, b)\ (x',\, b') \bullet E\ b\ b'\,]\,]$
by (*rule spec_fbI* [*rule_format*])
also have ...
$\sqsubseteq$
$[\,[F] \gg ([\,\lambda(x{::}\alpha,\, a) \bullet A\ a\,/\,\lambda(x{::}\alpha,\, b)\ (x',\, b') \bullet E\ b\ b'\,])\,]$
by (*intro monotonic_operators*, *rule coerceI*)
also have ...
$\sqsubseteq$
$[\,[F] \gg [\,F \wedge (\lambda(x{::}\alpha,\, a) \bullet A\ a)\,/\,\lambda(x{::}\alpha,\, b)\ (x',\, b') \bullet E\ b\ b'\,]\,]$
by (*rule monotonic_operators*, *rule assumeS*)

This style of development turns out to be a safe application of miracles because the fixed point properties of the feedback components mean that these coercions can eventually be eliminated using the following refinement rule.

lemma *fb_coerceE*: $\dfrac{[\,[E] \gg \langle f \rangle\,]}{[\,\langle f \rangle\,]}\quad \forall x\ a \bullet \text{fst}\ (f\ (x,\, a)) = x \Rightarrow E\ (x,\, a)$
by (*auto*)

Thus, once the dataflow element has been refined to an implementation (function), any coercions on the feedback components as inputs that have been used to aid that development can be eliminated, provided they are in fact established on the feedback components as outputs.

4 Input/output machines with feedback

The astute reader will have noted (perhaps with some annoyance) that we have not yet mentioned any concrete model of process behaviour which would legitimately allow us to consider the above formalism a refinement calculus for process networks. The paper has been so presented in order to stress the fact that all of the refinement calculus mechanisms of network composition are independent of the concrete process model to be adopted. Thus we are free to fit the refinement calculus approach onto virtually any (state-based) process model which supports any or all of the sequential, parallel and feedback forms of hookup.

Of course, in order to present any interesting examples in the use of the refinement calculus, it is necessary to choose a particular concrete process behaviour model. The first decision in choosing a model is determining how best to represent observations of the system components and an important part of this is deciding on an appropriate model for time. In the large informal case studies we have done in network refinement we have generally been concerned with physical flows such as water levels and line voltages [13, 15, 17, 18] and have made use of the real numbers to model time. Unfortunately real-analysis support in Isabelle is not really mature enough to be used for giving the sort of simple examples in refinement that we wish to present here. Instead we model system observables using the natural numbers as our model of time.

Another important decision concerning such a model is the selection of a class of specifications which may be considered terminal points of the design process, that is to say the process *implementations*. In the program refinement calculus the assignment statements (deterministic, total specifications) are used as the basis of the implementation language, the general class of implementations then being the closure of the assignments under the various program operators. The introduction of the feedback operator complicates this approach by having the potential to introduce non-determinism and even magic, even when applied to deterministic processes. We must be careful how feedback is used in the implementation language if we are to ensure the functionality of all implementations. Of course, a comparable problem also exists in the case of the while operator, the difference being that a while statement may introduce incompleteness rather than nondeterminism or inconsistency. Our approach to treating this problem is to construct an implementation language in such a way as to ensure that feedback loops do not introduce nondeterminism or inconsistency. An alternative approach might have been to follow the lead of the while-loop and introduce notions of partial-correctness (all fixed points are refinements) and total-correctness (there is exactly one fixed-point).

The rest of this section is devoted to presenting a simple model of multi-threaded computation based on digital traces and an implementation language which we call IFO machines, input/output machines with feedback.

4.1 Traces

Temporal observations of I/O machines are modeled by *traces*. The simple traces are functions from the natural numbers to instantaneous observations of the inputs or outputs of the I/O machine. We write Σ^* for the simple traces over Σ. More complex traces are built up as tuples of simple traces. We find it convenient to adopt a subscript notation for indexing of trace elements, for example writing t_n for the n^{th} element of the trace t.

Complex traces are composed/decomposed using the *tzip*/*tunzip* functions.

defs

tzip_def: $tzip \mathrel{\hat=} \lambda t \bullet (\lambda n \bullet ((\mathrm{fst}\ t)_n, (\mathrm{snd}\ t)_n))$

tunzip_def: $tunzip \mathrel{\hat=} \lambda t \bullet (\lambda n \bullet \mathrm{fst}\ t_n, \lambda n \bullet \mathrm{snd}\ t_n)$

Values may be attached to the front of simple traces.

defs *tcons_def*: $a\#t \mathrel{\hat{=}} \lambda n \bullet \textbf{if } n = 0 \textbf{ then } a \textbf{ else } t_{n-1} \textbf{ fi}$

An indexed collection of predicates/relations can be lifted to a trace predicate/relation by conjoining their pointwise applications.

defs *dpprod_def*: $\Pi\ \Phi \mathrel{\hat{=}} \lambda t \bullet \forall\, n \bullet \Phi_n\ t_n$

In theory, the distributed product operator offers the power to express any desired predicate/relation over traces. This is because every trace t has a corresponding *characteristic* predicate $\chi\ t$ which identifies exactly the given trace. The characteristic predicate of a trace may be expressed as the distributed product of the trace's elements

$$\chi\ t = \Pi n \bullet \lambda s \bullet s = t_n$$

and hence any predicate/relation may be characterised as a disjunction of distributed products.

$$\phi = \vee t \,|\, \phi\ t \bullet \chi\ t$$

For the purposes of this paper this theoretical completeness is sufficient, but it should be noted that in practice some form of sophisticated temporal logic would be convenient for expressing and reasoning about the properties of traces.

Trace predicates that may be expressed as distributed products or finite disjunctions of distributed products are what Alpern and Schneider [2] refer to as *safety* properties. These are characterised formally as: those predicates for which, any trace excluded by the predicate has a finite prefix, that has no extensions that satisfy the predicate. The dual notion of *liveness* is also introduced, a liveness property being one for which, every finite trace has an extension that satisfies the predicate. Informally, safety properties may be violated in a finite time, while liveness properties may not. These notions of safety and liveness have taken a central role in the search for tractable reasoning systems for distributed networks. A special form of safety property is the *invariant* which may be expressed as the distributed product of a constant function, that is in the form $(\Pi n \bullet I)$ for some instantaneous predicate I.

4.2 IFO machine constructors

The basis of the IFO machine language is the *I/O dynamic*, which consists of a predicate transformer of instantaneous states applied pointwise to an entire trace. In order to define this notion of "pointwise" application, the distributed product operator is lifted to processes in much the same way that the binary product was lifted.

defs
dfprod_def: $\Pi\ f \mathrel{\hat{=}} \lambda t\ n \bullet f_n\ t_n$
drprod_def: $\Pi\ R \mathrel{\hat{=}} \lambda t\ s \bullet (\forall\, n \bullet R_n\ t_n\ s_n)$
dprod_def: $\Pi\ p \mathrel{\hat{=}} \lambda \phi \bullet (\bigvee\ \Phi \,|\, (\Pi n \bullet \Phi_n) \Rrightarrow \phi\ \bullet\ (\Pi n \bullet p_n\ \Phi_n))$

The distributed product operator builds processes which calculate output traces by iteratively applying sequential programs pointwise to their input traces. Actually the distributed product operator is somewhat too general to be considered a real process operator, since it allows the sequential programs to vary with time. In order to realise this a process would need to have some innate sense of absolute time, where in actuality processes are only able to gauge the passage of time through the explicit use of devices such as clocks and counters. In light of this observation, we introduce a restricted form of distributed product (**do _ od**) in which the iterating program may not vary with the passage of time. This operator we call the *dynamic* program, or the do-loop, and presented in graphical form as an oval shaped network element as shown in Fig. 6(a).

defs *dynamic_def*: **do** p **od** $\mathrel{\hat{=}} \Pi n \bullet p$

While a complete approach to developing process networks would require the adoption of a particular sequential programing language for expressing dynamic programs, for the purposes of this paper it is adequate to abstract away from such details.

The basic introduction rule for dynamics allows a trace specification involving an invariant assumption and an invariant effect to be refined to a dynamic specification.

(a) dynamic (b) latch

Fig. 6. IO process elements.

lemma *spec_dynI*: $\dfrac{[\Pi n \bullet A \,/\, \Pi n \bullet E]}{\mathbf{do}\,[A \,/\, E]\,\mathbf{od}}$

proof (*simp*)

It is sufficient to observe that every trace property achieved by an invariant effect is satisfied by achieving the effect at every point in the trace.

fix ϕ *a* **assume** *a1*: $\forall n{::}\mathbb{N} \bullet A\ a_n$ **and** *a2*: $\forall b \bullet (\forall n{::}\mathbb{N} \bullet E\ a_n\ b_n) \Rightarrow \phi\ b$
show $\exists \Phi \bullet (\forall b \bullet (\forall n{::}\mathbb{N} \bullet \Phi_n\ b_n) \Rightarrow \phi\ b) \wedge (\forall n\ b \bullet E\ a_n\ b \Rightarrow \Phi_n\ b)$
by (*simp add*: *a2*)
qed

For the most part the observations we made in regard of the strong decoupling required by the product introduction rule apply again to this rule. An interesting point in this case is the fact that the decoupling appears to restrict us to the treatment of invariant properties. In fact, the restriction merely introduces a requirement to refine the trace specification by weakening assumptions and strengthening effects to the point where it is expressed by some invariant behaviour. Arbitrary trace properties may be freely used at any point up to the introduction of a dynamic design. The restriction at this point is not artificial, but rather a natural result of adopting the dynamic as the basic computational device. Of necessity, a dynamic can only effect behaviour that can be achieved by repeatedly performing the same (invariant) task. Nor is the choice of the dynamic particularly unfortunate or artificial, the majority of embedded, control and communications applications adopt exactly this architecture of tightly scheduled repetitive behaviour.

The main advantage in adopting dynamics as the basis for a network implementation language is the observation that deterministic dynamic processes are *causal*. We don't attempt to define this notion in the general case, but in the deterministic case a causal process is one for which the first *n* inputs uniquely determine the first *n* outputs. That is, the process does not look forward in time when determining the current output. Clearly this is a necessary requirement for any notion of process implementation.

Any process constructed from causal network elements using sequential and parallel hookup will also be causal, but, unfortunately, causality is not necessarily preserved under feedback, nor is it sufficient to preserve determinism under feedback. In order to preserve causality and determinism, we follow Stølen [28] and introduce the stronger notion of *guardedness*. A (deterministic) process is guarded if and only if the $(n{+}1)^{\text{th}}$ output is uniquely determined by the first *n* inputs. For a detailed discussion of guarded trace functions the reader is directed to Stølen [28].

The guardedness of a process may be ensured by introducing a delay or *latch* element into the network. Latches are depicted in network graphs as small triangles, as shown in Fig. 6(b).

defs *latch_def*: $a\triangleright \mathrel{\widehat{=}} (\lambda t \bullet a\#t)$

A feedback loop is causal if the enclosed (causal) process is guarded in the feedback component. This may be ensured by restricting feedback loops to be of the form

$$[\ p \gg (a\triangleright) \times \mathbf{1}\ ]$$

where *p* is a causal process. We call such feedback loops guarded-feedback loops and write

$$[\, a \longleftarrow p \,]$$

as shorthand for the above process.

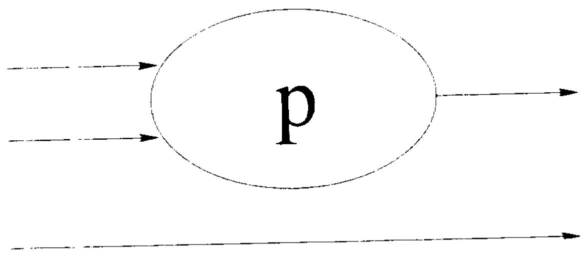


Fig. 7. Elided network connectives.

In addition to dynamics and latches, the causal processes **1**, ⟨*tzip*⟩ and ⟨*tunzip*⟩ are useful network connectives. We introduce abbreviations ⊗ and ⊗ for the processes ⟨*tzip*⟩ and ⟨*tunzip*⟩ respectively. We do not explicitly represent these processes on network graphs, since it is straightforward to infer their presence from the way in which explicit network elements are connected. For example, since a dynamic must have a single input stream and a product must be between processes, if can be inferred that the diagram of Fig. 7 refers to the process

$$(\otimes \gg \mathbf{do}\, p\, \mathbf{od}) \times \mathbf{1}.$$

We are now in a position to define the IFO network language.

- A dynamic machine $\mathbf{do}\langle f\rangle\mathbf{od}$ is an IFO machine for any function f.
- The processes **1**, ⊗ and ⊗ are IFO machines.
- If p and q are IFO machines then so are $p \gg q$ and $p \times q$.
- If p is an IFO machine then so is $[a \longleftarrow p]$.

In the case that p is of the form

$$\otimes\gg \mathbf{do}\langle g\rangle\mathbf{od}\gg \otimes$$

it is straightforward to show that

$$[a \longleftarrow p]$$

uniquely constructs the feedback trace x which satisfies the recursive equations

$$x_0 = a$$
$$(x_{n+1}, t_n) = g(x_n, s_n),\ n \in \mathbb{N}$$

where s and t are respectively the global input and output. Thus, guarded feedbacks ensure the preservation of both determinism and causality and all IFO machines are both deterministic and causal.

5 Some simple examples

In this section we present example refinements which illustrate the basic application of the network refinement calculus.

5.1 Accumulating a sum

We begin with a straightforward, but thorough exercising of the refinement calculus approach. In order to improve the readability of the example, we elide lambda bindings representing state variables, under the convention that the names of the state variables and the context identify the formal arguments to the lambda abstraction. For example, given state variables x and y, we would write $x = y$ for the relational abstraction $\lambda x\ y \bullet x = y$.

A simple network application is to calculate a running sum of the values being passed along input s, passing the results on output t. This may be expressed very straightforwardly as a relational specification.

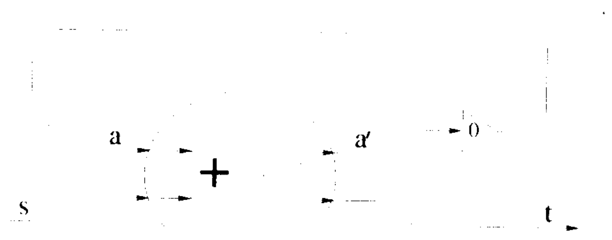


Fig. 8. The summing machine.

let $?SUM = (\forall n \bullet t_n = \sum_{k=0}^{n} s_k)$

A simple solution to this problem is to use a feedback component a to store the previous value of the sum. The relationship between a and s is:

let $?PSUM = a_0 = 0 \wedge (\forall n \bullet a_{n+1} = \sum_{k=0}^{n} s_k)$

The desired property of t can then be effected by adding a to s at each cycle as shown in Fig. 8.
The refinement argument begins by introducing the feedback component.

have [true / *?SUM*]
$\sqsubseteq$
  [[true / *?SUM*]]
  by (*rule spec_fbI*)

Next we focus on the design of the internal component, first coercing the feedback property on the input side.

have [true / *?SUM*]
$\sqsubseteq$
  [*?PSUM*] $\gg$ ([*?PSUM* / *?SUM*])
 proof –*apply coerceI and assumeS* **qed**

Now we focus on refining the specification statement, requiring first that b be a latched copy of t and that t be the sum of a and s at each point.

have [*?PSUM* / *?SUM*]
$\sqsubseteq$
  [*?PSUM* / $b = (0\#t) \wedge (\forall n \bullet t_n = a_n + s_n)$]
 proof (*rule commitS* [*rule_format*])

The entailment is easily demonstrated by induction.

 qed

The next step is to decompose the commitment relation, introducing relational sequencing and product, then lifting these to the process level.

also have ...
$\sqsubseteq$
  [*?PSUM* /
    $(b = t \wedge (\forall n \bullet t_n = a_n + s_n))$
    $\gg ((b = 0\#a) \times (t = s))$]
  (**is** _ $\sqsubseteq$ [_ / *?SELT* $\gg$ _])
 by (*rule commitS* [*rule_format*], *auto*)
also have ...
$\sqsubseteq$
  [*?PSUM* / *?SELT*] $\gg$ ((*0* $\triangleright$) $\times$ **1**)

proof *–apply midI and spec_prod* **qed**

Focusing on the summing element, we begin by eliminating the assumption.

have [*?PSUM* / *?SELT*]

$\sqsubseteq$

[true / *?SELT*]

by (*rule assumeW* [*rule_format*], *auto*)

Since both b and t depend on both a and s, we zip them up to allow a dynamic implementation to use them both. We write *as* for the zipping of a and s and *bt* for the zipping of b and t.

also have ...

$\sqsubseteq$

$\otimes \gg$ [true / *?SELT* (*tunzip as*) (*tunzip bt*)] $\gg \oslash$

by (*rule spec_zipI*)

Focusing on the dynamic element, we re-express the commitment as an relational invariant, so as to implement it as a dynamic.

have [true / *?SELT* (*tunzip as*) (*tunzip bt*)]

$\sqsubseteq$

[true / $\Pi n \bullet (b = a + s \land t = a + s)$]

proof *–apply commitS* **qed**

also have ...

$\sqsubseteq$

do $\langle (a', t) := (a + s, a + s) \rangle$ **od**

proof *–apply spec_dynI* **qed**

Thus, we have shown that the design of Fig. 8 achieves the required commitment, provided that the feedback component properly stores the partial sums.

finally have [true / *?SUM*]

$\sqsubseteq$

([*?PSUM*] $\gg \otimes \gg$ **do** $\langle (a', t) := (a+s, a+s) \rangle$ **od** $\gg \oslash) \gg ((0 \triangleright) \times \mathbf{1})$.

The final step is then to eliminate the feedback input coercion, by demonstrating that the feedback component does store the partial sums.

also have ...

$\sqsubseteq$

[$0 \longleftarrow \otimes \gg$ **do** $\langle (a', t) := (a+s, a+s) \rangle$ **od** $\gg \oslash$]

proof *–convert body to single function, then apply fb_coerceE* **qed**

Thus we are now left with a pure IFO implementation of the summing machine specification.

finally have [true / ($\forall n \bullet t\ n = \sum_{k=0}^{n} s\ k$)]

$\sqsubseteq$

[$0 \longleftarrow \otimes \gg$ **do** $\langle (a', t) := (a+s, a+s) \rangle$ **od** $\gg \oslash$] .

5.2 The magic of refinement

Our second example examines the potential dangers involved in introducing feedback assumptions and also the protections built in to the refinement approach.

The main fear with making use of feedback assumptions is the possibility that the assumptions may become self-fulfilling prophecies. Since an implementation need only achieve its commitment when its inputs satisfy the necessary assumptions, might it not be possible to use the postulated feedback assumptions to construct an implementation which achieves its commitment only because it has been assumed to do so?

For example, consider the following refinement sequence.

have $[\,\mathrm{true} \,/\, \lambda s\ t \bullet A\ s\ t\,]$

$\sqsubseteq$

$[\,[\lambda(x, s) \bullet A\ s\ x] \gg [\ \lambda(x, s) \bullet A\ s\ x \,/\, \lambda(x, s)\ (x', t) \bullet A\ s\ x \wedge A\ s\ t\,]\,]$

proof *–apply spec_fbI, coerceI, assumeS, and commitS* **qed**

At this point in the design process we have set up a potentially dangerous situation. Since x is already assumed to satisfy the required property $A\ s\ x$, we can get a refinement of the specification statement by simply copying the input x across to the outputs x' and t.

also have ...

$\sqsubseteq$

$[\,[\lambda(x, s) \bullet A\ s\ x] \gg \langle \lambda(x, s) \bullet (x, x) \rangle\,]$

proof *–apply commitS, then funI* **qed**

The design process seems to have gone completely wrong! It is clear that any further development from this point could not possibly result in an implementation of the original specification, but what could have gone wrong? We have carefully made use of trusted refinement laws and what we have arrived at is a design which we know cannot achieve the desired result. Is there a problem with the definition of feedback, or worse still a problem with the notion of refinement?

Neither of these. What has happened is that we have made a bad design choice, a choice which has led us to a complete dead end. It is not just that we can no longer get to an implementation that satisfies the specification, it is no longer possible to get to any implementation at all. In order to get code from the above design we must eliminate the input coercion and this we can no longer do. This coercion is our protection against making incorrect use of the feedback assumptions it allows us to introduce. It can only be removed when we have been able to establish the required effect on the outputs absolutely and without recourse to any feedback assumptions. This is not a unique phenomena in the refinement calculus, there are many situations where we are allowed the freedom to introduce designs that have no implementation. The basic argument in favour of risking such unsafe decisions is that the work required to ensure a design decision is safe is comparable to the work required to construct an implementation. In most cases, it is far better to risk the occasional deadend refinement sequence, than to be continually put to the work of checking that every design decision is safe.

That said, there are situations in which this is not the case, primarily in the development of large-scale projects in which multiple developers and indeed teams of developers are involved. Postponing the elimination of the input coercion until the final stage of design becomes a considerable risk when there are numbers of developers responsible for the various subcomponents of a system. In such cases it is imperative to have a method of eliminating the coercion before the commencement of separate development efforts on subcomponents. Fortunately it is possible to introduce variants on *spec_fbI* and *fb_coerceE* which support a more localised approach to dealing with feedback assumptions.

lemma *spec_fbI*$_2$: $\dfrac{[\,A \,/\, E\,]}{\{A\} \gg [\,[\,\lambda(x, a) \bullet A\ a \,/\, \lambda(x, a)\ (x', b) \bullet E\ a\ b\,]\,]}$

by *(simp)*

lemma *fb_coerceE*$_2$:

$$\frac{\{A\} \gg\ [\,[E] \gg [\,\lambda(x, a) \bullet A\ a \,/\, F\,]\,]}{[\,[\,\lambda(x, a) \bullet A\ a \,/\, F\,]\,]} \quad (\lambda(x, s) \bullet A\ s \wedge (\exists t \bullet F\ (x, s)\ (x, t))) \Rrightarrow E$$

by *(simp, blast)*

The important aspect of the variant feedback introduction rule *spec_fbI*$_2$ is that the environment assumption is retained external to the feedback loop. The variant coercion elimination rule *fb_coerceE*$_2$ then makes use of this external assertion to allow the input coercion to be eliminated once the designer has finished making use of it in the design of the feedback component. Thus the coercion is eliminated at a more appropriate time, prior to the commencement of subcomponent development efforts.

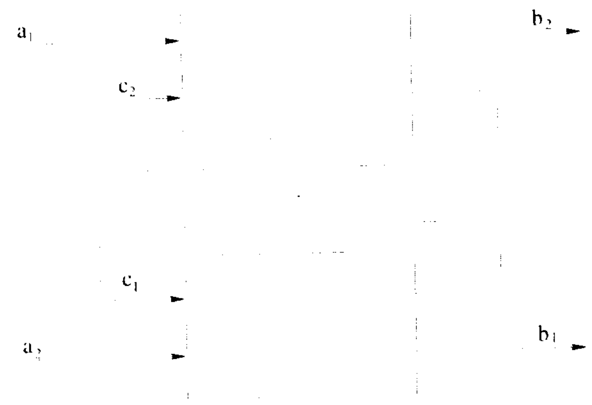


Fig. 9. Using separate calculating units.

5.3 A liveness example

Our final example explores the use of liveness properties as feedback assumptions. Even though the refinement rules for introducing feedback assumptions do not distinguish between liveness and safety, the importance this distinction has played in previous formalisms seems to demand that some attention be paid to the matter. However, we do feel justified in only considering a simple and somewhat contrived example of their use.

A stream processing machine accepts intermittent communications of natural numbers on a channel a_1. For each communication on channel a_1 it must calculate the value f_1 $(f_2\ a_{1m})$ and output it on channel b_1. Since calculating the functions f_1 and f_2 may take a number of cycles, we simply require that for every communication on a_1 there is eventually a response on b_1. A similar requirement is placed on a second channel a_2, with the exceptions that the processing functions are applied in the opposite order and that outputs are placed on a second channel b_2.

In formalising these requirements, we model the notion of an intermittent channel as a trace over the natural numbers extended with a null element $\perp$ $(\mathbb{N}_\perp)$. Thus a trace of an intermittent channel consists of natural numbers intermixed with null communications. We write δa_n for the requirement that the n^{th} communication on the intermittent channel a is a proper communication. Formally, we express the *eventual* calculation of some function f by the following specification.

let $?CALC\ f = \lambda(a::(\mathbb{N}_\perp)^*)\ (b::(\mathbb{N}_\perp)^*) \bullet$
$(\forall\, n \bullet \delta a_n \Rightarrow (\exists\, m \bullet n \leq m \wedge b_m = f\ a_n))$

The requirement on b_1 is then

$$?CALC\ (f_1 \circ f_2)\ a_1\ b_1$$

and that on b_2 is

$$?CALC\ (f_2 \circ f_1)\ a_2\ b_2.$$

Both these requirements are liveness properties, because they do not dictate a deadline for when the calculations must be finished. Of course, it is not really intended that arbitrarily long times be allowed for the calculations. Rather this is a convenient way of leaving the determination of the calculation times until later in the design process, when more is known about the mechanics of the calculations.

The symmetry of the problem suggests the possibility of decomposing the process into one element for calculating f_1 and one for calculating f_2. The idea would be to send inputs on a_1 directly to the f_2 processing element, then to the f_1 processing element and *vice versa* for inputs on a_2. The resulting process topology is depicted in Fig. 9.

The first step is then to introduce feedback components for passing these intermediate calculations between the two processing elements. We retain the original assumption, so as to allow the early elimination of the feedback coercion as described in the previous section.

have [true / $?CALC\ (f_1 \circ f_2)\ a_1\ b_1 \wedge ?CALC\ (f_2 \circ f_1)\ a_2\ b_2$]
$\sqsubseteq$
  {true} $\gg$
  [[true / $?CALC\ (f_1 \circ f_2)\ a_1\ b_1 \wedge ?CALC\ (f_2 \circ f_1)\ a_2\ b_2$]]
 proof –*apply spec_fbI*$_2$ **qed**

The strategy for using these intermediate channels is to perform f_2 processing on messages from a_1, then pass them along c_1 for f_1 processing before outputing them on b_1. Similarly c_2 is used for intermediate calculations of messages on the a_2 channel.

This strategy is implemented by introducing the intended properties of the intermediate channels as feedback assumptions, then using these properties to show that the proposed behaviour of the output channels satisfies the original requirements.

also have ...
$\sqsubseteq$
  {true} $\gg$
  [[$?CALC\ f_2\ a_1\ c_1 \wedge ?CALC\ f_1\ a_2\ c_2$] $\gg$
    [$?CALC\ f_2\ a_1\ c_1 \wedge ?CALC\ f_1\ a_2\ c_2$ /
      $?CALC\ (f_1 \circ f_2)\ a_1\ b_1 \wedge ?CALC\ (f_2 \circ f_1)\ a_2\ b_2$]]
 proof –*apply coerceI and assumeS* **qed**
also have ...
$\sqsubseteq$
  {true} $\gg$
  [[$?CALC\ f_2\ a_1\ c_1 \wedge ?CALC\ f_1\ a_2\ c_2$] $\gg$
    [$?CALC\ f_2\ a_1\ c_1 \wedge ?CALC\ f_1\ a_2\ c_2$ /
      $?CALC\ f_2\ a_1\ c_1{}' \wedge ?CALC\ f_1\ a_2\ c_2{}' \wedge$
      $?CALC\ f_1\ c_1\ b_1 \wedge ?CALC\ f_2\ c_2\ b_2$]]
 proof (*intro monotonic_operators, rule commitS* [*rule_format*])

Eventually calculating one function and then eventually calculating the other is equivalent to eventually calculating the composition of the two functions.

 qed

The last step in this design is to remove the feedback coercions that we have now finished making use of.

also have ...
$\sqsubseteq$
  {true} $\gg$
  [[$?CALC\ f_2\ a_1\ c_1 \wedge ?CALC\ f_1\ a_2\ c_2$] $\gg$
    [true /
      $?CALC\ f_2\ a_1\ c_1{}' \wedge ?CALC\ f_1\ a_2\ c_2{}' \wedge$
      $?CALC\ f_1\ c_1\ b_1 \wedge ?CALC\ f_2\ c_2\ b_2$]]
 by (*intro monotonic_operators, rule assumeW* [*rule_format*], *auto*)
also have ...
$\sqsubseteq$
  [[true /
      $?CALC\ f_2\ a_1\ c_1{}' \wedge ?CALC\ f_1\ a_2\ c_2{}' \wedge$
      $?CALC\ f_1\ c_1\ b_1 \wedge ?CALC\ f_2\ c_2\ b_2$]]
 by (*rule fb_coerceE*$_2$ [*rule_format*], *auto*)
also have ...
$\sqsubseteq$
  [[true /
      $?CALC\ f_2\ a_1\ c_1{}' \wedge ?CALC\ f_1\ a_2\ c_2{}' \wedge$

$?CALC\ f_1\ c_1\ b_1 \wedge ?CALC\ f_2\ c_2\ b_2\,]]$
by (*intro monotonic_operators*, *rule assumeW* [*rule_format*], *auto*)

The next step in the development would be to decompose the internal specification in parallel, but we leave that as an exercise for the interested reader. The purpose of this section was to demonstrate the use of liveness properties as feedback assumptions, in this case the properties $?CALC\ f_2\ a_1\ c_1$ and $?CALC\ f_1\ a_2\ c_2$. The result has been a fairly banal repetition of our previous treatments of feedback assumptions. Indeed, the most novel aspect of the design derivation was the utilisation of the "localised" elimination rule *fb_coerceE*$_2$ that was introduced in Sect. 5.2. This is not surprising, since the introduction rules for feedback coercions make no distinction between liveness and safety properties.

6 Conclusions

This paper has considered the application of the refinement calculus to the specification and design of process networks consisting of sequential, parallel and feedback elements. The sequential and parallel hookup operators are well known from the literature [3, 5], but to the best of our knowledge the predicate transformer feedback operator is a novel generalisation of the relational operator proposed by Katis *et al* [12].

In order to make use of feedback assumptions in process developments, we have suggested the novel use of coercions on the feedback components as inputs. We have shown how these magical annotations may be eliminated from the final implementation (*fb_coerceE*) or even from earlier stages of the design in the case of large-scale developments (*fb_coerceE*$_2$). The addition of these refinement laws makes the refinement calculus a powerful tool for analysing process networks, capable of treating both safety and liveness properties with equal simplicity.

An important aspect of the resulting network refinement-calculus is its abstraction from any underlying model of process computation. The network refinement-calculus is potentially able to support a wide range of implementation models either in isolation or even in combination.

As a means of providing concrete examples of the network refinement-calculus in action, we have introduced the IFO machine as a simple, abstract model of process computation. The basic building block of the IFO language is the *dynamic*, a novel, if straightforward, generalisation of the product operator. Specialised refinement rules have been introduced to support IFO implementations and some examples given in the refinement of IFO machines. Again we stress that the refinement calculus is not restricted in its application to these IFO machines. It could just as easily have been applied to dataflow/stream-processing functions, real-time automaton, state machines, or indeed any state-based model of process computation. Applying the network refinement-calculus to event-based models may be more problematic since they sometimes lack a clear distinction between input and output components.

It is worth noting here that the various hookup introduction rules make no distinction between the use of safety and liveness properties, either in assumptions or in commitments. In fact we had no need of introducing the notions at all until we came to the introduction rule for IFO dynamics. The treatment of liveness has generally been a problematic aspect of network design formalisms. Misra and Chandy [20] restricted their approach entirely to safety properties, while Lamport's TLA [1] finesses the problem by introducing the arcane notion of the closure of a liveness property to overcome a prohibition on the use of liveness properties as feedback assumptions. More recently, Stølen has proposed a feedback verification rule which allows the use of liveness properties, but once again it involves the difficult computation of closures and in addition the introduction of an invariant property (with a safety component). All of these difficulties stem from the treatment of feedback through the sophisticated notion of recursion, rather than through the more naive notion of fixed-point. Although at the function level, these notions are almost identical, at the relational level they differ vastly. A recursion-based view of feedback immediately places the focus on finite traces and safety, forcing an indirect treatment of infinite traces and hence liveness. By adopting the naive fixed-point approach suggested by Katis *et al* [12], we have rendered the distinction between safety and liveness irrelevant. The resulting feedback-hookup introduction rule is vastly simpler, involving no calculations at all.

Apart from the gains made in adopting the naive fixed-point view of feedback, the adoption of predicate-transformer semantics offers significant gains in tractability over pure relational semantics. A significant strength of predicate-transformer semantics is the ability to provide a clear, semantic separation between assumptions and commitments. While the assumption and commitment associated with a specification statement are semantically unique, any given relational specification admits quite arbitrary assumption/commitment decompositions which must be treated through the introduction of adaptation rules. The availability of (distinct) assertions and coercions also greatly adds to the power and flexibility of the refinement calculus, as compellingly demonstrated by our approach to the introduction and elimination of feedback assumptions. By introducing feedback properties as coercions we flag an intention to *make* the feedback property true. If assertions were used, it would instead indicate that we were assuming that they were *already* true. Such a distinction could not be made in a pure relational model.

A number of future research directions are suggested by the issues raised in this paper. The interface between trace-based design and sequential program code needs considerable elaboration. It would certainly be useful to be able to introduce more general program operators than the simple nonterminating loop of *spec_dynI*. This might allow, for example, the clear decomposition of a program into initialisation and processing phases. An important future direction of this work is also to apply the refinement calculus to truly real-time processes. In order to do this a computation model based on continuous functions of real-time [16] may be adopted. However, it will require considerable development of the Isabelle real-analysis environment to make this feasible. Work in this direction is currently underway in collaboration with the SVRC at the University of Queensland.

Acknowledgements

The work reported in this paper has been supported in part by Australian Research Council grant number A4913006: *Formal methods for the specification and refinement of time-based systems and processes.*

This paper has been has been a long time in the gestation and the author wishes to acknowledge valuable discussions with (in temporal order) Ian Hayes, Colin Millerchip, Keith Duddy, Colin Fidge, Tony Dekker, Lindsay Groves, Tony Cant, Jim McCarthy, Jamie Shield, and Clare Mahony. Thanks also is due to the feedback (sic) received from anonymous reviewers of earlier versions of this paper.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
2. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.
3. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer, 1998.
4. R. J. R. Back. A calculus of program derivations. *Acta Informatica*, 25:593–624, 1988.
5. R. J. R. Back and M. J. Butler. Exploring summation and product operators in the refinement calculus. In *Mathematics of Program Construction*, volume 947 of *LNCS*, pages 128–158. Springer-Verlag, 1995.
6. R. J. R. Back and M. J. Butler. Fusion and simultaneous execution in the refinement calculus. *Acta Informatica*, 35(11), 1998.
7. T. Cant, J. McCarthy, and B. Mahony. Dove: Design oriented verification and evaluation. `http://www.dsto.defence.gov.au/esrl/itd/dove/index.html`.
8. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, New Jersey, 1976.
9. R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science.* American Mathematical Society, 1967.
10. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), October 1969.
11. C. B. Jones. The search for tractable ways of reasoning about programs. Technical Report Series UMCS-92-4-4, Department of Computer Science, University of Manchester, 1992.
12. P. Katis, N. Sabadini, and R.F.C. Walters. Span(graph): A categorical algebra of transition systems. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, volume 1349 of *LNCS*. Springer, 1997.
13. B. P. Mahony. *The Specification and Refinement of Timed Processes.* PhD thesis, University of Queensland, 1991. `ftp://ftp.it.uq.edu.au/pub/Thesis/brendan_mahony.ps.Z`.
14. B. P. Mahony. Calculating the least conjunctive refinement and promotion in the refinement calculus. *Formal Aspects of Computing*, 11:75–105, 1999.

15. B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A central heater. In J. M. Morris and R. C. Shaw, editors, *Proceedings of the* 4^{th} *Refinement Workshop*, Workshops in Computing, pages 138–149. Springer-Verlag, 1991.
16. B. P. Mahony and I. J. Hayes. Using continuous real functions to model timed histories. In *Proceedings of the 6th Australian Software Engineering Conference (ASWEC91)*, 1991.
17. B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering*, 18(9), September 1992.
18. B. P. Mahony, C. Millerchip, and I. J. Hayes. A boiler control system: A case study in timed refinement. In *International Invitational Workshop - Design and Review of Software Controlled Safety-Related Systems*, Ottawa, June 1993.
19. C. Martin. *Preordered Categories and Predicate Transformers.* PhD thesis, Oxford University Computing Laboratory, Programming Research Group, 1991.
20. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.
21. C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988. Reprinted in [22].
22. C. C. Morgan and T. Vickers, editors. *On the Refinement Calculus.* Springer-Verlag, 1992.
23. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
24. G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Language and Systems*, 11(4):517–561, 1989.
25. T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle's logics: HOL. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2002/doc/logics-HOL.pdf`.
26. P. K. Pandya and M. Joseph. P-A logic - a compositional proof system for distributed programs. *Distributed Computing*, 5(1):37–54, 1991.
27. K. Stolen, F. Dederichs, and R. Weber. Specification and refinement of networks of asynchronously communicating agents using the assumption/commitment paradigm. *Formal Aspects of Computing*, 8, 1996.
28. K. Stølen. Assumption/commitment rules for dataflow networks—with an emphasis on completeness. In Hanne Riis Nielson, editor, *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming*, volume 1058 of *LNCS*, pages 356–372. Springer, 1996.
29. A. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.
30. M. Wenzel. The Isabelle/Isar reference manual. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2002/doc/isar-ref.pdf`.
31. J. Zwiers, J. Coenen, and W. P. de Roever. A note on compositional refinement. In 5^{th} *Refinement Workshop.* Spinger-Verlag, 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** August 2002	**3. REPORT TYPE AND DATES COVERED** Conference Publication

4. TITLE AND SUBTITLE Theorem Proving in Higher Order Logics	**5. FUNDING NUMBERS** 727-01-22-01
6. AUTHOR(S) Edited by Víctor A. Carreño, César A. Muñoz, and Sofiène Tahar	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER** L-18211
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** NASA/CP-2002-211736

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified–Unlimited Subject Category 61 Distribution: Standard Availability: NASA CASI (301) 621-0390	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 words)***

The TPHOLs International Conference serves as a venue for the presentation of work in theorem proving in higher-order logics and related areas in deduction, formal specification, software and hardware verification, and other applications. Fourteen papers were submitted to Track B (Work in Progress), which are included in this volume. Authors of Track B papers gave short introductory talks that were followed by an open poster session. The FCM 2002 Workshop aimed to bring together researchers working on the formalisation of continuous mathematics in theorem proving systems with those needing such libraries for their applications. Many of the major higher order theorem proving systems now have a formalisation of the real numbers and various levels of real analysis support. This work is of interest in a number of application areas, such as formal methods development for hardware and software application and computer supported mathematics. The FCM 2002 consisted of three papers, presented by their authors at the workshop venue, and one invited talk.

14. SUBJECT TERMS Theorem proving; Formal methods; Reasoning; Verification; Logic; Deduction; Specification	**15. NUMBER OF PAGES** 198
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

Author Index