ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**Mary Sheeran, Chalmers Technical University, Sweden, and Satnam Singh,
University of Glasgow, UK (Eds)**

Designing Correct Circuits

Proceedings of the 3rd Workshop on Designing Correct Circuits
(DCC96), Båstad, Sweden, 2-4 September 1996

ISBN: 3-540-76102-0

Paper:

Ensuring Correctness of Ruby Transformations

Ole Rasmussen

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Ensuring Correctness of Ruby Transformations

Ole Rasmussen

Dept. of Information Technology, Technical University of Denmark

DK-2800 Lyngby, Email: `osr@it.dtu.dk`

Abstract

Equivalence transformations are widely used in practical designs of VLSI circuits using Ruby. This paper demonstrates how proofs of these equivalences easily may be performed within a formal framework by a theorem prover. The proof tool used is called RubyZF and contains a semantical embedding of Ruby within Zermelo-Fraenkel set theory using the Isabelle theorem prover. The use of the system is exemplified by a concrete example taken from the T-Ruby design system.

1 Introduction

A common design method using the relation based language Ruby [3], is the so called *design by calculation*. A design is derived from an abstract specification in Ruby by applying a number of correctness preserving transformations until a design, only expressed in terms of implementable relations, is reached. This method is widely used in work with Ruby and is demonstrated in for example [4, 10, 8, 2]. The transformations are equalities or conditional equalities expressing general facts about relations and their combination and are typically reused from one design to an other. To support this style of design a tool called T-Ruby has been constructed, based on a formalisation of Ruby as a strongly dependently typed language of functions and relations [12]. The T-Ruby system enables the user to perform the desired transformations in the course of a design and ensures that the transformations are correctly applied. However, in T-Ruby the actual transformations cannot be proved and we have therefore constructed a proof tool, called RubyZF, containing a *conservative semantical embedding* of Ruby within the Isabelle theorem prover [6] using a formulation of Zermelo-Fraenkel set theory (ZF). The development of RubyZF serves three purposes in connection with T-Ruby: to give Ruby a machine verified semantics; to prove general transformation rules for inclusion in T-Ruby's database; and to prove conditions and conjectured rewrite rules originating from a concrete series of transformations used in a design. Naturally, RubyZF may in itself serve as a platform for further Ruby developments as e.g. performing proofs of various refinement

In Ruby, the VLSI circuits are described by binary relations of their behaviour. The class of circuits we are interested in describing are combinational and single-clocked sequential circuits together with their combination. The definition of Ruby used in T-Ruby and RubyZF is based on the so-called Pure Ruby subset as introduced by Rossen [9]. This makes use of the observation that the above class of circuits may be expressed in terms of four basic elements: two relations and two combining forms, which are usually defined in terms of synchronous streams of data. All other circuits and combinators in this work are defined solely in terms of Pure Ruby elements. Consequently, we have a very simple basic framework, which helps us to ensure the soundness of the complete system. It is of major importance that the definitions used in RubyZF correspond to the ones used in T-Ruby and vice versa. The development of the two systems therefore go hand in hand, thus many of the definitions presented here originate from the T-Ruby system (and further back from Rossen) and likewise some definitions in T-Ruby come from this work.

The Ruby transformation rules in the literature are seldomly formally proved and their correctness are often claimed from vague graphical arguments. This may lead to incorrect rules or to rules where detailed preconditions are left out or not properly formulated. This paper demonstrates how proofs of the transformation rules commonly used in connection with Ruby designs may easily be performed within the formal framework of RubyZF. This is often regarded as a tedious process, but by using the high-level proof tools from Isabelle and by tailoring the system specifically to Ruby proofs, a high degree of automation may be achieved.

After an introduction of Isabelle and RubyZF, three types of transformation proofs are presented: equalities with simple combinators, equalities with recursive, and equalities proved by structural induction over Pure Ruby. The last section is based on a concrete design example from T-Ruby and demonstrates how new specific combinators may easily be defined and new transformation rules proved.

2 Isabelle

Isabelle is a generic logical framework meant for defining different proof systems and is presented thoroughly in [6]. The user must distinguish between two levels of abstraction, the *meta-level* and the *object-level*, where the former is used to define a particular object logic. The basic Isabelle system defines the meta logic, which is a fragment of intuitionistic high-order logic, and the meta language, which is a simply typed lambda calculus.

Isabelle is implemented in Standard ML (SML) and the proof commands are SML functions changing the current proof state. The major proof method is backward proof applying *tactics* to the current proof state. The main tactics apply lists of rules to a subgoal using various forms of resolution. Tactics can be composed into new more complex tactics using *tacticals*, which are higher-order SML functions.

The implication, $\phi \implies \psi$, expresses logical entailment and nested implication, $\phi_1 \implies (\phi_2 \cdots (\phi_n \implies \psi))$ is conveniently written as $\llbracket \phi_1 ; \cdots ; \phi_n \rrbracket \implies \psi$. Universal quantification, $\bigwedge x \cdot \phi$, expresses generality and means that ϕ is true for all x . Nested quantifications are written $\bigwedge x_1 \cdots x_n \cdot \phi$. Universal quantifications at the outermost level may be left out.

The distribution of Isabelle includes an implementation of Zermelo-Fraenkel set theory built as an extension to classical first-order logic. A large number of theories for basic mathematics already exist in the standard distribution of ZF and are introduced in [5].

Three generic packages have been used extensively in this work: the *simplifier* performs conditional and unconditional rewriting using contextual information and a given set of rewrite rules; the *classical* reasoning package provides a number of tactics to prove theorems in the style of the sequent calculus (e.g. `fast_tac`); and the *inductive* definition package permits the formalisation of all monotone inductive definitions and is based on a fixedpoint approach.

2.1 Types in RubyZF

In ZF types are modelled as sets of values, which means that there is no distinction between types and terms, as they both denote sets. Dependent types are easily modelled as parameterised sets and subtypes come directly as subsets. For readers familiar with HOL-like systems, a major difference is that there is no specific machinery to help handling types and no implicit type polymorphism, so in many cases type parameters must be supplied explicitly.

Type conditions appear in the proof state as subgoals to the main proof and there is no distinction between type-goals and other goals. However, a number of specialised tactics have been developed in RubyZF to recognise and solve goals related to type checking automatically. For each new constructor defined, we prove a type rule and store it in an internal list. The main tactic accesses the information in that list and tries to solve all type goals in the proof state as much as possible and leave unsolvable goals to the user. Furthermore, type checking is incorporated into the conventional resolution tactics to perform type checking after each resolution step. In this way we obtain automatic type checking and type goals does only rarely pop up to be solved manually (typically trivial arithmetic goals). Special typed versions of the classical tactics are provided to perform type checking in connection with classical proof steps. The typed version of, for example `fast_tac`, is called `fast_tac_t`.

3 Ruby in RubyZF

This section gives a brief introduction to the formalisation of Ruby in Isabelle ZF. The complete development is described in [7]. First a theory of signals is introduced modelling the synchronous streams of data. Then the four elements of Pure Ruby are defined and a Ruby relation type introduced. Finally, a number of circuits and combinators are defined in terms of Pure Ruby.

3.1 Signals

The synchronous streams, called signals, are represented as a function $\mathbb{Z} \rightarrow \mathcal{T}$ (written $\text{sig}(\mathcal{T})$), where $\mathcal{T}$ is the type of values and $\mathbb{Z}$ represents the *time*. $\mathcal{T}$ ranges over the possible channel types, ChTy , and when reasoning about Ruby we are interested in making a distinction between three kinds of channel types: base types, pairing of types, and a list of a type. More formally signals can be expressed as:

$$\begin{aligned} \text{sig} &= \mathbb{Z} \rightarrow \text{ChTy} \\ \text{ChTy} &= \text{BasChTy} \mid \text{ChTy} \times \text{ChTy} \mid \text{nlist}[n]\text{ChTy} \end{aligned}$$

where $\text{nlist}[n]\alpha$ are lists of length n and elements of type α . Since nlists are parameterised in n , Ruby relations may have *dependent product* types. The base types, BasChTy , will typically be natural numbers, bits etc. but no explicit restriction is made.

In RubyZF we first define some basic types for relations and signals as depicted in the left side of Figure 1. The relation type, $\alpha \sim \beta$, is defined as the powerset of pairs of type $\alpha \times \beta$. Time is modelled as integers and a signal, $\text{sig}(\alpha)$, as a function from time to α . Finally, relations between signals of type α and β have the type $\alpha \overset{\text{sig}}{\sim} \beta$.

$$\begin{aligned} \alpha \sim \beta &\equiv \wp(\alpha \times \beta) & \langle a \# b \rangle &\equiv \lambda t \in \text{time} \cdot \langle a't, b't \rangle \\ \text{sig}(\alpha) &\equiv \text{time} \rightarrow \alpha & \text{snil} &\equiv \lambda t \in \text{time} \cdot \text{nnil} \\ \alpha \overset{\text{sig}}{\sim} \beta &\equiv \text{sig}(\alpha) \sim \text{sig}(\beta) & [l \overset{n}{\frown} a] &\equiv \lambda t \in \text{time} \cdot \text{nsnoc}_n(l't, a't) \end{aligned}$$

Figure 1: Definition of signal types and constructors

Ruby relations are all binary relations on single signals and therefore a number of constructors are defined to compose complex signals into a single signal. Figure 1 defines three constructor functions for signals by abstracting their corresponding basic constructors over time. Signal pairing, $\langle a \# b \rangle$, pairs two signals a and b . The two last make use of a theory defining nlists, which provides the usual operations such as: nnil , the nil-element and nsnoc , to concatenate an element to the back of a list. Using this, snil constructs a signal from nnil , and $[l \overset{n}{\frown} a]$ concatenates a signal a to the back of a signal list l of length n . In the figure above $\cdot$ denotes the explicit operator for functional application.

3.2 Pure Ruby

The four Pure Ruby elements are formalised in ZF as appropriate subsets of type $\alpha \overset{\text{sig}}{\sim} \beta$ and their definitions are shown in Figure 2. Viewed as relations, $\text{spread}(r)$ is the lifting to signals of the pointwise relation described by r . The delay element, $\mathcal{D}$, relates a signal to another signal which has an offset of one time tick, $(R ; S)$ describes relational composition and $[R, S]$ relational product. In the figure the set $\text{dtyp}(R)$ defines, if R is a signal relation of type $R \in \alpha \overset{\text{sig}}{\sim} \beta$,

$$\begin{aligned} \text{spread}(r) &\equiv \{ \langle x, y \rangle \in \text{sig}(\text{domain}(r)) \times \text{sig}(\text{range}(r)) \mid \forall t \in \text{time} \cdot \langle x't, y't \rangle \in r \} \\ \mathcal{D}_\alpha &\equiv \{ \langle x, y \rangle \in \text{sig}(\alpha) \times \text{sig}(\alpha) \mid \forall t \in \text{time} \cdot x't = y'(t + 1) \} \\ R ; S &\equiv \{ \langle x, z \rangle \in \text{domain}(R) \times \text{range}(S) \mid \exists y \cdot \langle x, y \rangle \in R \wedge \langle y, z \rangle \in S \} \\ [R, S] &\equiv \{ \langle x, y \rangle \in \text{sig}(\text{dtyp}(R) \times \text{dtyp}(S)) \times \text{sig}(\text{rtyp}(R) \times \text{rtyp}(S)) \mid \\ &\quad \langle \text{pri}(x), \text{pri}(y) \rangle \in R \wedge \langle \text{sec}(x), \text{sec}(y) \rangle \in S \} \end{aligned}$$

Figure 2: The definition of Pure Ruby in ZF

a subset of α where $\text{sig}(\text{dtyp}(R))$ contains all the elements of the domain of R . Correspondingly the set $\text{rtyp}(R)$ is defined for the range. The two functions pri and sec are destructor functions for the first and the second part of signal pairs respectively. For spread , serial and parallel composition the bounding set types can be inferred from the relational arguments by means of domain and range functions or alternatively the functions dtyp and rtyp in the case of parallel composition. This is unfortunately not possible for the delay element, so the type must be given as an explicit parameter.

Viewed as circuits, $\text{spread}(r)$ describes the synchronously clocked combinational circuit with the functionality of r . $\mathcal{D}$ describes the basic sequential circuit (a latch), $(R ; S)$ the sequential composition of two circuits and $[R, S]$ the parallel composition.

Modelling time as integers means that our Ruby time is discrete supporting the view that the time models the clock ticks in a synchronous circuit. The use of integers, which complicates the matter in a theorem prover, is needed to have a double infinite time base. Consequently, the time has no start or end point and it is assumed to have run and will run forever. This enable us, for example, to prove the so called retiming property for Ruby circuits as will be explained in Section 4.3. Others [2, 1] has modelled time as natural numbers which is considerably easier, but then they cannot prove retiming etc.

Figure 3: Graphical interpretations of the four Pure Ruby elements

A feature of Ruby is that relations and combinators have a natural *graphical interpretation*, corresponding to an abstract floorplan for the circuits which they describe. The conventional graphical interpretation of spread is as a labelled rectangular box, where the number of wire stubs reflects the types of the relations in an obvious manner. The components of the domain are drawn up the left hand side and the components of the range up the right. The remaining elements of Pure Ruby are drawn in an intuitively obvious way, as illustrated in Figure 3.

For each of the four elements of Pure Ruby, appropriate type, introduction and elimination rules are proved. For the spread element the following rules are proved in one step by `fast_tac`:

$$\begin{array}{ll}
 \text{spread_type} & r \in \alpha \sim \beta \implies \text{spread}(r) \in \alpha \overset{\text{sig}}{\sim} \beta \\
 \text{spreadI} & \llbracket \forall t \in \text{time}. \langle x^t, y^t \rangle \in r ; r \in \alpha \sim \beta ; x \in \text{sig}(\alpha) ; y \in \text{sig}(\beta) \rrbracket \implies \\
 & \langle x, y \rangle \in \text{spread}(r) \\
 \text{spreadE} & \llbracket \langle x, y \rangle \in \text{spread}(r) ; \llbracket \forall t \in \text{time} \cdot \langle x^t, y^t \rangle \in r \rrbracket \implies P \rrbracket \implies P
 \end{array}$$

To allow proofs by structural induction over the four Pure Ruby elements, the Pure Ruby subset has been characterised by an inductive definition. This defines the set $\alpha \overset{\text{pure}}{\sim} \beta$ as the subset of $\alpha \overset{\text{sig}}{\sim} \beta$, which contains all relations constructed from Pure Ruby elements whose domain type is $\text{sig}(\alpha)$ and range type is $\text{sig}(\beta)$. For example, for spread this leads to the following Ruby type rule:

$$\text{spreadR} \quad \llbracket \alpha \in \text{ChTy} ; \beta \in \text{ChTy} ; r \in \alpha \sim \beta \rrbracket \implies \text{spread}(r) \in \alpha \overset{\text{pure}}{\sim} \beta$$

where ChTy contains all BasChTy and is closed under pairing and nlist 's. From the inductive definition it is possible to prove the induction theorem for Pure Ruby:

$$\begin{array}{l}
 \text{ruby_induct} \quad \llbracket R \in \alpha \overset{\text{pure}}{\sim} \beta ; \\
 \quad \bigwedge r \alpha \beta \cdot r \in \alpha \sim \beta \implies P(\alpha, \beta, \text{spread}(r)) ; \\
 \quad \bigwedge \alpha \cdot P(\alpha, \alpha, \mathcal{D}_\alpha) ; \\
 \quad \bigwedge R S \alpha \beta \gamma \cdot \llbracket R \in \alpha \overset{\text{pure}}{\sim} \beta ; S \in \beta \overset{\text{pure}}{\sim} \gamma ; \\
 \quad \quad P(\alpha, \beta, R) ; P(\beta, \gamma, S) \rrbracket \implies P(\alpha, \gamma, R ; S) ; \\
 \quad \bigwedge R S \alpha_1 \alpha_2 \beta_1 \beta_2 \cdot \llbracket R \in \alpha_1 \overset{\text{pure}}{\sim} \beta_1 ; S \in \alpha_2 \overset{\text{pure}}{\sim} \beta_2 ; \\
 \quad \quad P(\alpha_1, \beta_1, R) ; P(\alpha_2, \beta_2, S) \rrbracket \implies P(\alpha_1 \times \alpha_2, \beta_1 \times \beta_2, [R, S]) \\
 \rrbracket \implies P(\alpha, \beta, R)
 \end{array}$$

All relations in this paper belong to $\overset{\text{pure}}{\sim}$ and all combinators construct Pure Ruby relations from Pure Ruby arguments.

3.3 Circuits and Combinators

The circuits and combinators are defined solely in terms of Pure Ruby. Circuits are non-parameterised signal relations and the combinators are parameterised signal relations typically combining signal relations into new signal relations. A special class of circuits only regrouping signals are called wiring relations.

The definitions of some wiring relations used in this paper are shown in Figure 4. The relation ι is the polymorphic identity relation. reorg describes the conversion between the two possibilities of pairing three elements. cross relates a pair of values to the reversed pair. dub relates a signal to two copies of the same signal thus describing a conventional fork. π_1 and π_2 are projection relations which relate a pair to its first and second component respectively. Furthermore two wiring relations are used in connection with nlists . The relation NNIL relates two empty signal nlists of any type and the relation apr_n (append right) relates a list of n elements a_1 and an element a_2 to a list of $n + 1$ elements where a_2 is concatenated to the back of a_1 . Note that the identifier ab in the definitions is used in place of pattern matching.

$$\begin{aligned}
\iota_\alpha &\equiv \text{spread}(\{ab \in \alpha \times \alpha \mid \exists a \cdot ab = \langle a, a \rangle\}) \\
\text{reorg}_{\alpha\beta\gamma} &\equiv \text{spread}(\{ab \in ((\alpha \times \beta) \times \gamma) \times (\alpha \times (\beta \times \gamma)) \mid \\
&\quad \exists a \ b \ c \cdot ab = \langle \langle a, b \rangle, c \rangle, \langle a, \langle b, c \rangle \rangle\}) \\
\text{cross}_{\alpha\beta} &\equiv \text{spread}(\{ab \in ((\alpha \times \beta) \times (\beta \times \alpha)) \mid \exists a \ b \cdot ab = \langle \langle a, b \rangle, \langle b, a \rangle \rangle\}) \\
\text{dub}_\alpha &\equiv \text{spread}(\{ab \in (\alpha \times (\alpha \times \alpha)) \mid \exists a \cdot ab = \langle a, \langle a, a \rangle \rangle\}) \\
\pi_{1\alpha\beta} &\equiv \text{spread}(\{ab \in ((\alpha \times \beta) \times \alpha) \mid \exists a \ b \cdot ab = \langle \langle a, b \rangle, a \rangle\}) \\
\text{NNIL} &\equiv \text{spread}(\{\langle \text{nnil}, \text{nnil} \rangle\}) \\
\text{apr}_{\alpha n} &\equiv \text{spread}(\{\langle \langle a_1, a_2 \rangle, b \rangle \in (\text{nlist}[n]\alpha \times \alpha) \times \text{nlist}[\text{succ}(n)]\alpha \mid \\
&\quad b = \text{nsnoc}_n(a_1, a_2)\})
\end{aligned}$$

Figure 4: Definitions of some wiring relations

Examples of combinator definitions are shown in Figure 5. Fst produces the relational product of the argument and the identity relation (a similar combinator Snd produces the opposite product). The combinator $R \Downarrow S$, called *below*,

$$\begin{aligned}
\text{Fst}_\gamma(R) &\equiv [R, \iota_\gamma] \\
\text{below}'_{\alpha\beta\gamma\delta\epsilon\zeta\eta}(R, S) &\equiv \text{reorg}_{\alpha\epsilon\zeta} ; \text{Snd}_\alpha(S) ; \text{reorg}_{\alpha\beta\eta}^{-1} ; \text{Fst}_\eta(R) ; \text{reorg}_{\gamma\delta\eta} \\
R \Downarrow S &\equiv \text{below}'_{(\text{ddtyp}(R), \text{rdtyp}(R), \text{drtyp}(R), \text{rrtyp}(R), \text{ddtyp}(S), \text{rdtyp}(S), \text{rrtyp}(S))}(R, S) \\
\text{powf}_{\alpha n}(R) &\equiv \text{rec}(n, \iota_\alpha, \lambda x \ y \cdot y ; R^x) \\
\text{mapf}'_{\alpha\beta n}(F) &\equiv \text{rec}(n, \text{NNIL}, \lambda x \ y \cdot \text{apr}_{\alpha x}^{-1} ; [y, F^x] ; \text{apr}_{\beta x}) \\
\text{mapf}_n(F) &\equiv \text{mapf}'_{(\text{dtyp}(\bigcup \text{range}(F)), \text{rtyp}(\bigcup \text{range}(F)), n)}(F) \\
\text{tri}_{\alpha n}(R) &\equiv \text{mapf}_n(\lambda x \in \text{nat} \cdot \text{pow}_{\alpha x}(R)) \\
\text{colf}'_{\alpha\beta\gamma n}(F) &\equiv \text{rec}(n, \text{Fst}_\beta(\text{NNIL}) ; \text{cross}_{(\text{nlist}[0]\gamma, \beta)}, \\
&\quad \lambda x \ y \cdot \text{Fst}_\beta(\text{apr}_{\alpha x})^{-1} ; (y \Downarrow F^x) ; \text{Snd}_\beta(\text{apr}_{\gamma x})) \\
\text{colf}_{\beta n}(F) &\equiv \text{colf}'_{(\text{ddtyp}(\bigcup \text{range}(F)), \beta, \text{rrtyp}(\bigcup \text{range}(F)), n)}(F) \\
\text{rowf}_{\beta n}(F) &\equiv (\text{colf}_{\beta n}(\lambda x \in \text{nat} \cdot (F^x)^{-1}))^{-1} \\
\text{map}_n(R) &\equiv \text{mapf}_n(\lambda x \in \text{nat} \cdot R)
\end{aligned}$$

Figure 5: Definitions of some combinators

describes partial composition for two four sided relations, where the second component in the domain of R is connected to the first component of the range of S (in the definition $^{-1}$ denotes relational inverse which is also defined in terms of Pure Ruby). Repeated structures are traditionally described in Ruby by homogeneous combinators. Here we define more general combinators (suffixed with an f , where the argument is a function from its position in the structure to a signal relation such that we can describe heterogeneous structures. Repeated serial and parallel composition are defined

by `powf` and `mapf` respectively. Repeated serial and parallel composition may also be combined to describe a triangle structure, `tri`. Finally, the combinators, `colf` and `rowf`, are examples of combinators taking relations between signal pairs as argument and producing column and row structures respectively. Note that for most combinators a fully type parameterised version is first defined and then used to define a notational shorthand, where the types are deduced from the relational (or functional) arguments. Non-dependent versions of the recursive combinators (the combinator `map` shown as an example in the figure) are easily defined and Isabelle automatically uses the non-dependent versions if the argument is independent of the position in the structure. The graphical interpretations of some of the combinators are shown in Figure 6.

Figure 6: Graphical interpretation of some combinators

For each circuit and combinator the same suite of rules is defined: two type rules, an introduction, and an elimination rule. One type rule states that the relation belongs to the general signal relation type and the other that the relation belongs to the set of Pure Ruby relations given appropriate arguments. The introduction and elimination rules follow the same pattern as we saw for the Pure Ruby constructors. Below the Ruby type rule for `mapf` shows that $\text{mapf}_n(F)$ is a Pure Ruby relation if F is a function to a Pure Ruby relation:

$$\text{mapfR} \quad [n \in \text{nat} ; F \in \text{nat} \rightarrow \alpha \stackrel{\text{pure}}{\sim} \beta] \implies \text{mapf}_n(F) \in \text{nlist}[n] \alpha \stackrel{\text{pure}}{\sim} \text{nlist}[n] \beta$$

Furthermore for combinators equality rules are proved expressing that the given definition actually corresponds to a more conventional definition using data values. The two equality rules `mapf_zero` and `mapf_succ` express that `mapf` enjoys the expected mathematical properties in terms of data values.

$$\begin{aligned} \text{mapf_zero} \quad & \langle \text{snil}, \text{snil} \rangle \in \text{mapf}_0(F) \\ \text{mapf_succ} \quad & [n \in \text{nat} ; F \in \text{nat} \rightarrow \alpha \stackrel{\text{sig}}{\sim} \beta ; a \in \text{sig}(\alpha) ; b \in \text{sig}(\beta) ; \\ & l_a \in \text{sig}(\text{nlist}[n] \alpha) ; l_b \in \text{sig}(\text{nlist}[n] \beta)] \implies \\ & \langle [l_a \hat{\sim} a], [l_b \hat{\sim} b] \rangle \in \text{mapf}_{\text{succ}(n)}(F) \Leftrightarrow (\langle a, b \rangle \in F^n \wedge \langle l_a, l_b \rangle \in \text{mapf}_n(F)) \end{aligned}$$

4 Proof Examples

The basic Ruby relations and their combining forms generate a relational algebra which defines a large number of equivalences. These are used in the practical design process with Ruby usually performed in the T-Ruby system. One of the main purposes of RubyZF is to prove these equality rules to ensure the correctness of a design in T-Ruby. In this section we demonstrate how different classes of equality rules quite easily can be proved in RubyZF and how special tactics can be developed to support and automate this process. The proof examples are divided into three main classes: simple equalities, recursive equalities and transformations proved by structural induction. The last section shows proofs of some specific transformation rules originating from a concrete design example performed in T-Ruby.

4.1 Simple Equality Proofs

To prove equality rules concerning simple combinators automatically, the tactic `ProveSimp` has been defined. It splits the equality into two implications adding appropriate data values to the relations. Then it solves the two implications by the typed version of the classical reasoning tactic `fast_tac`, which uses the rule set `RubyZF_CS` containing all

introduction and elimination rules for the simple combinators. Furthermore type information is added to the internal type rule list. Examples of equations proved by above tactic and used in the comming examples are shown in Figure 7.

parcompdist	$\llbracket R \in \alpha_1 \stackrel{sig}{\sim} \beta_1 ; S \in \alpha_2 \stackrel{sig}{\sim} \beta_2 ; T \in \beta_1 \stackrel{sig}{\sim} \gamma_1 ; U \in \beta_2 \stackrel{sig}{\sim} \gamma_2 \rrbracket \Longrightarrow$ $[R, S] ; [T, W] = [(R ; T), (S ; W)]$
assoccomp	$\llbracket R \in \alpha \stackrel{sig}{\sim} \beta ; S \in \beta \stackrel{sig}{\sim} \gamma ; T \in \gamma \stackrel{sig}{\sim} \delta \rrbracket \Longrightarrow R ; S ; T = S ; (R ; T)$
compinv	$\llbracket R \in \alpha \stackrel{sig}{\sim} \beta ; S \in \beta \stackrel{sig}{\sim} \gamma \rrbracket \Longrightarrow (R ; S)^{-1} = S^{-1} ; R^{-1}$
invinv	$R \in \alpha \stackrel{sig}{\sim} \beta \Longrightarrow (R^{-1})^{-1} = R$
Id_compl	$R \in \alpha \stackrel{sig}{\sim} \beta \Longrightarrow \iota_\alpha ; R = R$
plid	$\pi_{1\alpha\beta}^{-1} ; \pi_{1\alpha\beta} = \iota_\alpha$
NNILinv	$NNIL^{-1} = NNIL$
crossover	$\llbracket R \in \alpha \stackrel{sig}{\sim} \beta ; S \in \gamma \stackrel{sig}{\sim} \delta \rrbracket \Longrightarrow [R, S] ; \text{cross}_{\beta\delta} = \text{cross}_{\alpha\gamma} ; [S, R]$
fstsndpar	$\llbracket R \in \alpha \stackrel{sig}{\sim} \beta ; S \in \gamma \stackrel{sig}{\sim} \delta \rrbracket \Longrightarrow \text{Fst}_\gamma(R) ; \text{Snd}_\beta(S) = [R, S]$
fstsndcomm	$\llbracket R \in \alpha \stackrel{sig}{\sim} \beta ; S \in \gamma \stackrel{sig}{\sim} \delta \rrbracket \Longrightarrow \text{Fst}_\gamma(R) ; \text{Snd}_\beta(S) = \text{Snd}_\alpha(S) ; \text{Fst}_\delta(R)$
belowmid	$\llbracket R \in \alpha \times \beta \stackrel{sig}{\sim} \gamma \times \delta ; S \in \epsilon \times \zeta \stackrel{sig}{\sim} \eta \times \theta ; X \in \gamma \stackrel{sig}{\sim} \zeta \rrbracket \Longrightarrow$ $S \Downarrow (R ; \text{Fst}_\delta(X)) = (\text{Snd}_\epsilon(X) ; S) \Downarrow R$
par_below	$\llbracket R \in \alpha \times \beta \stackrel{sig}{\sim} \gamma \times \delta ; T \in \epsilon \times \zeta \stackrel{sig}{\sim} \beta \times \eta ;$ $U \in \theta \stackrel{sig}{\sim} \alpha ; V \in \kappa \stackrel{sig}{\sim} \epsilon ; S \in \mu \stackrel{sig}{\sim} \zeta \rrbracket \Longrightarrow$ $[[U, V], S] ; (R \Downarrow T) = (\text{Fst}_\beta(U) ; R) \Downarrow ([V, S] ; T)$
duplicate	$\llbracket R \in \alpha \stackrel{sig}{\sim} \beta ; \text{function}(R) \rrbracket \Longrightarrow R ; \text{dub}_\beta = \text{dub}_\alpha ; [R, R]$

Figure 7: Examples of simple equality rules

The graphical interpretation of Ruby is very convenient to get the intuition behind a circuit, but it can sometimes, however, be hazardous to rely too much on the drawings. The equivalence `duplicate` on Figure 8 seems to hold when viewing the drawing, but it turns out that this is only the case under the assumption that R is functional. Adding this assumption `duplicate` can be proved by `ProveSimp`.

Figure 8: Graphical interpretation of duplicate

Equivalences can also be proved using the Isabelle simplifier. We have defined a tactic, `Rubysimp`, taking a list of equality rules as argument and additionally exploits the Ruby type information. Examples of rules proved by the simplifier are shown in Figure 9. Using the simplifier leads to much faster and smaller (in use of memory) proofs than

fstpardist	$\llbracket R \in \alpha \stackrel{sig}{\sim} \beta ; S \in \gamma \stackrel{sig}{\sim} \delta ; T \in \epsilon \stackrel{sig}{\sim} \alpha \rrbracket \Longrightarrow \text{Fst}_\gamma(T) ; [R, S] = [T ; R, S]$
sndpardist	$\llbracket R \in \alpha \stackrel{sig}{\sim} \beta ; S \in \gamma \stackrel{sig}{\sim} \delta ; T \in \epsilon \stackrel{sig}{\sim} \gamma \rrbracket \Longrightarrow \text{Snd}_\alpha(T) ; [R, S] = [R, T ; S]$

Figure 9: Examples of equality rules proved by the simplifier

using the classical tactics, both because of the higher abstraction level at which the simplifier proofs are performed (the classical tactics expand all combinators) and because of a more efficient implementation of the simplifier. However,

since the rewriting process is not always going in the same direction (as is the case of, for example, simplifying arithmetic expressions) no fixed simplification set can be developed and the simplification rules must be chosen manually for every proof step. Thus simplifier proofs are usually less automatic in contrast to the classical tactic, `ProveSimp`, which succeeds with no interaction.

4.2 Recursive Equality Proofs

Proofs of equality rules containing recursive combinators are performed by induction over the size of the combinators followed by a number of rewrite steps using simple equalities, such as those seen in the previous section. Usually during a simplification proof a specific equality between simple relations is needed, so a typical proof procedure is, that the major steps are governed by the simplifier while small trivial equivalences are proved by `ProveSimp`. Figure 10 shows a selection of recursive equality rules.

<code>pow_rev</code>	$\llbracket n \in \text{nat} ; R \in \alpha \stackrel{\text{sig}}{\sim} \alpha \rrbracket \implies \text{pow}_{\alpha n}(R) ; R = R ; \text{pow}_{\alpha n}(R)$
<code>pow_inv</code>	$\llbracket n \in \text{nat} ; R \in \alpha \stackrel{\text{sig}}{\sim} \alpha \rrbracket \implies \text{pow}_{\alpha n}(R)^{-1} = \text{pow}_{\alpha n}(R^{-1})$
<code>commcomp1</code>	$\llbracket n \in \text{nat} ; R \in \alpha \stackrel{\text{sig}}{\sim} \alpha ; S \in \alpha \stackrel{\text{sig}}{\sim} \alpha ; R ; S = S ; R \rrbracket \implies$ $R ; \text{pow}_{\alpha n}(S) = \text{pow}_{\alpha n}(S) ; R$
<code>mapf_inv</code>	$\llbracket F \in \text{nat} \rightarrow \alpha \stackrel{\text{sig}}{\sim} \beta ; n \in \text{nat} \rrbracket \implies$ $\text{mapf}_n(\lambda m \in \text{nat} . (F^i m)^{-1}) = (\text{mapf}_n(F))^{-1}$
<code>maptricom2</code>	$\llbracket R \in \alpha \stackrel{\text{sig}}{\sim} \alpha ; S \in \alpha \stackrel{\text{sig}}{\sim} \alpha ; n \in \text{nat} ; R ; S = S ; R \rrbracket \implies$ $\text{tri}_{\alpha n}(R) ; \text{map}_n(S) = \text{map}_n(S) ; \text{tri}_{\alpha n}(R)$
<code>rowmap1</code>	$\llbracket R \in \alpha \times \beta \stackrel{\text{sig}}{\sim} \gamma \times \alpha ; S \in \delta \stackrel{\text{sig}}{\sim} \beta ; n \in \text{nat} \rrbracket \implies$ $\text{Snd}_{\alpha}(\text{map}_n(S)) ; \text{row}_{\alpha n}(R) = \text{row}_{\alpha n}(\text{Snd}_{\alpha}(S) ; R)$

Figure 10: Examples of equality rules with recursive combinators

To demonstrate how these rules may be proved we show the concrete Isabelle proofs of the last rule of Figure 10, `rowmap1`. This should also reveal some of the problems encountered doing simplifier proofs with Ruby. To help performing simplifier proofs we have defined two rule lists: `recr1` containing expansion rules for the recursive combinators; and `invr1` containing rules to move inversion as much inside relations as possible e.g. `compinv`. The equality is entered into Isabelle:

```
- val prems = goal RecComb.thy
  "!!R. [| R : A*B<~>C*A; S:De<~>B; n:nat |] ==>
    Snd(A,Map(n,S)) ;; row(A,n,R) = row(A, n, Snd(A,S)); R )"
Level 0
1. !!R. [| R : A * B<~>C * A; S : De<~>B; n : nat |] ==>
  Snd(A, Map(n, S)) ;; row(A, n, R) = row(A, n, Snd(A, S)) ;; R
```

We do induction over n , leading to a base case and an inductive case:

```
- by (eresolve_tac [nat_induct] 1);
Level 1
1. !!R. [| R : A * B<~>C * A; S : De<~>B |] ==>
  Snd(A, Map(0, S)) ;; row(A, 0, R) = row(A, 0, Snd(A, S)) ;; R
2. !!R x. [| R : A * B<~>C * A; S : De<~>B; x : nat;
  Snd(A, Map(x, S)) ;; row(A, x, R) = row(A, x, Snd(A, S)) ;; R |] ==>
  Snd(A, Map(succ(x), S)) ;; row(A, succ(x), R) =
  row(A, succ(x), Snd(A, S)) ;; R
```

The base case can either be solved by the tactic `ProveSimp` or by simplification. We choose to solve it by the latter

using simple equalities:

```
- by (Rubysimp (recrl@invrl@[Fst_def,Snd_def,parcompdist,crossover,
NNIL_NNIL,Id_compl,assoccomp RS sym]) 1);
Level 2
1. !!R x.[ | R : A * B<~>C * A; S : De<~>B; x : nat;
    Snd(A, Map(x, S)) ;; row(A, x, R) = row(A, x, Snd(A, S) ;; R) | ] ==>
    Snd(A, Map(succ(x), S)) ;; row(A, succ(x), R) =
    row(A, succ(x), Snd(A, S) ;; R)
```

where 'RS sym' tells the simplifier to use the equality in the right-left direction. The number of rules applied in the above proof step illustrates that although the simplifier it-self is automatic, it may require some effort by the user to find all the correct rules needed in a specific case. For the remaining subgoal, the inductive case, we expand the recursive combinators and pull relations inside the Snd combinator using sndcompdist:

```
- by (Rubysimp (recrl@[sndcompdist RS sym,assoccomp RS sym]) 1);
Level 3
1. !!R x.[ | R : A * B<~>C * A; S : De<~>B; x : nat;
    Snd(A, Map(x, S)) ;; row(A, x, R) = row(A, x, Snd(A, S) ;; R) | ] ==>
    Snd(A, apr(De, x)~ ;; [[Map(x, S),S]] ;; apr(B, x) ;;
    apr(B, x)~) ;; row(A, x, R) -- R ;; Fst(A, apr(C, x)) =
    Snd(A, apr(De, x)~) ;;
    row(A, x, Snd(A, S) ;; R) -- (Snd(A, S) ;; R) ;; Fst(A, apr(C, x))
```

We can now reduce the successive combination of apr_n and apr_n^{-1} to the identity by apr_aprinv and then remove the identity by the rule Id_compr . We split the serial combination inside the Snd combinator and reach:

```
- by (Rubysimp [apr_aprinv,Id_compr,assoccomp,sndcompdist] 1);
Level 4
1. !!R x.[ | R : A * B<~>C * A; S : De<~>B; x : nat;
    Snd(A, Map(x, S)) ;; row(A, x, R) = row(A, x, Snd(A, S) ;; R) | ] ==>
    Snd(A, apr(De, x)~) ;; (Snd(A, [[Map(x, S),S]]) ;;
    (row(A, x, R) -- R ;; Fst(A, apr(C, x)))) =
    Snd(A, apr(De, x)~) ;;
    (row(A, x, Snd(A, S) ;; R) -- (Snd(A, S) ;; R) ;; Fst(A, apr(C, x)))
```

This can be solved by the rule besidepar , followed by exploiting the induction hypothesis.

```
- by (Rubysimp [besidepar,assoccomp RS sym] 1);
Level 5
!!R. [ | R : A * B<~>C * A; S : De<~>B; n : nat | ] ==>
    Snd(A, Map(n, S)) ;; row(A, n, R) = row(A, n, Snd(A, S) ;; R)
No subgoals!
```

The example clearly demonstrates why simplification proofs in Ruby can be very difficult to automate more than to a certain level. First the simplification cannot use a static simplification set since rewrite rules are typically used in both directions. Secondly the associativity of serial composition is exploited all the time in both directions to get desired rules to match. A final obstacle in simplifier proofs is that the user interactively must decide which rules to use. This can be quite tedious and therefore a proof process usually consists of several applications of the simplifier until the correct set of rules has been found or extra lemmas proved by Provesimp . Some help can be achieved by defining rule lists containing rules which all perform similar steps e.g. as shown above with the lists invrl and recrl .

4.3 Retiming

Timing properties of circuits described with Pure Ruby are modelled using the delay element. A special feature of Ruby is that both timing and combinational transformations can be described with the same kind of equality rules. The rules we have seen so far express facts about general signal relations and thus also hold for the delay element. However, in this section we employ the sequential properties of the delay element to prove some specific timing properties of Pure Ruby.

Circuits described in terms of Pure Ruby relations certainly have a state due to the delay element. However, their behaviour do not depend on the *absolute* time and they are therefore said to be *timeless*. In this section we prove that

$$\begin{array}{ll}
\text{delay_Id} & \mathcal{D}_\alpha ; \mathcal{D}_\alpha^{-1} = \text{id}_\alpha \\
\text{D_invD_lemma} & t \in \text{int} \implies a^t t = (\lambda x \in \text{int} \cdot a^t(x-1))^{t+1} \\
\text{retime_spread} & r \in \alpha \sim \beta \implies \text{retime}_{\alpha\beta}(\text{spread}(r))
\end{array}$$

Figure 11: Timing related equivalences

all Pure Ruby relations are timeless and therefore enjoy the so called *retiming* property, which is expressed as:

$$\text{retime}_{\alpha\beta}(R) \equiv R = (\mathcal{D}_\alpha ; R ; \mathcal{D}_\beta^{-1})$$

Retiming is very useful in connection with the development of systolic circuits and in the next section we will see an application. That the retiming property holds for all Pure Ruby relations can be expressed by:

$$\text{retime } R \in \alpha \sim^{\text{pure}} \beta \implies \text{retime}_{\alpha\beta}(R)$$

This may be proved by structural induction over the four Pure Ruby elements. We enter the theorem into Isabelle and perform structural induction using `ruby_induct`:

```

- val prems = goal Retime.thy "!!R.R:A<R>B ==> retime(A,B,R)";
- by (eresolve_tac [ruby_induct] 1);
Level 1
1. !!R r A B. r : A~B ==> retime(A, B, spread(r))
2. !!R A. retime(A, A, D(A))
3. !!R Ra S A B C.
   [| Ra : A<R>B; S : B<R>C; retime(A, B, Ra); retime(B, C, S) |] ==>
   retime(A, C, Ra ;; S)
4. !!R Ra S A1 A2 B1 B2.
   [| Ra : A1<R>B1; S : A2<R>B2; retime(A1, B1, Ra); retime(A2, B2, S) |] ==>
   retime(A1 * A2, B1 * B2, [[Ra,S]])

```

Clearly this leads to four new subgoals, stating that the retiming property must hold for each Pure Ruby element. The three last subgoals are easily proved by simplification using the theorem `delay_Id` (and its reverse $\mathcal{D}_\alpha^{-1} ; \mathcal{D}_\alpha = \text{id}_\alpha$). It expresses that combined delays and inverse delays (anti delays) cancel out each other. This is provable by simplification of integer expressions and is a consequence of the decision of modelling time as integers. If we had modelled time as natural numbers and thus had a singly infinite time, the theorem could not be proved. Some timing related equivalences are given in Figure 11. The first subgoal above still remains and expresses that the spread element is timeless as stated by `retime_spread`. The proof is performed in RubyZF as follows:

```

- val prems = goalw Retime.thy [retime_def]
  "!!r.r:A~B==> retime(A,B,spread(r))";
Level 0
1. !!r. r:A~B ==> spread(r) = D(A) ;; spread(r) ;; D(B)~

```

We split the equality into two implications and type the attached data values:

```

- by (prove_equal prems 1);
Level 1
1. !!r x y.[| r:A~B; <x, y>:spread(r); x:sig(A); y:sig(B) |] ==>
   <x, y>:D(A) ;; spread(r) ;; D(B)~
2. !!r x y.[| r:A~B; <x,y>:D(A) ;; spread(r) ;; D(B)~; x:sig(A); y:sig(B) |]==>
   <x, y>:spread(r)

```

We use the classical tactic `safe_step_tac_t` a number of times on subgoal 1, mainly to apply elimination rules:

```

- by (REPEAT (safe_step_tac_t Ruby_cs prems 1));
Level 2
1. !!r x y.[| r:A~B; x:sig(A); y:sig(B); ALL t:time. <x't, y't>:r |] ==>
   <x, y>:D(A) ;; spread(r) ;; D(B)~
2. !!r x y.[| r:A~B; <x,y>:D(A) ;; spread(r) ;; D(B)~; x:sig(A); y:sig(B) |]==>
   <x, y>:spread(r)

```

Next we resolve with the Ruby introduction rules and the lemma `D_invD_lemma` expressing a simple fact about integers. This is used to get proper instantiations of the existential quantified variables originating from the serial compo-

sition; i.e. an instantiation of the two signals in between the delays on each side of the spread element.

```
- by (SELECT_GOAL (REPEAT (SOMEGOAL
  (resolve_tac_c [] ([D_invD_lemma]@RubyI))))1);
Level 3
1. !!r x y t. [| r:A~B; x:sig(A); y:sig(B); ALL t:time.<x't,y't>:r; t:time |]==>
  <(lam xa:time. x'(xa $- $# 1))'t, (lam x:time. y'(x $- $# 1))'t> : r
2. !!r x y. [| r:A~B; <x,y>:D(A) ;; spread(r) ;; D(B)~; x:sig(A); y:sig(B) |]==>
  <x, y>:spread(r)
```

where $\$-$ denotes integer minus and $\$ \#$ denotes conversion from natural numbers to integers. Solving subgoal 1 is now only a matter of integer simplification:

```
- by (asm_simp_tac integ_ss 1);
Level 4
1. !!r x y. [| r:A~B; <x,y>:D(A) ;; spread(r) ;; D(B)~; x:sig(A); y:sig(B) |]==>
  <x, y>:spread(r)
```

The final goal can be solved by a combination of `fast_tac_t` and simplification adding the integer simplification set using `addss`. This enables `fast_tac_t` to perform simplification after each classical step.

```
- by (fast_tac_t (Ruby_cs addss integ_ss) prems 1);
Level 5
!!r. r:A~B ==> retime(A, B, spread(r))
No subgoals!
```

4.4 Convolution

In T-Ruby the Ruby language is not regarded as static and therefore new combinators are often defined in connection with specific designs. This constitutes no problem in RubyZF as the basic implementation of Ruby forms a good platform for such enhancements. T-Ruby has been used to develop a design for a VLSI circuit for 2-dimensional discrete convolution which is described in [11]. In this section we do not show the actual circuit development in T-Ruby, but how the new combinators easily can be defined in terms of existing ones and specific transformation rules used in the design may be proved.

4.4.1 Definition of New Combinators

Two new combinators, `rdrf` and `fork`, are introduced in the convolution example. The combinator `rdrf` describes a kind of column structure and may be defined in terms of `colf` and the projection relation π_1 , as shown on the left side of Figure 12. The right part of the figure shows how `forkn`, an n -way fork, may be defined using the row combinator. Their definitions in RubyZF are:

$$\begin{aligned} \text{rdrf}_{\beta n}(F) &\equiv \text{colf}_{\beta, n}(\lambda m \in \text{nat} \cdot (F'm ; \pi_1^{-1}_{(\beta, \text{nat})})) ; \pi_1(\beta, \text{nlist}[n] \text{nat}) \\ \text{fork}_{\alpha n} &\equiv \pi_1^{-1}_{(\alpha, \text{nlist}[n] \text{nat})} ; \text{row}_{\alpha n}(\pi_1(\alpha, \text{nat}) ; \text{dub}_{\alpha}) ; \pi_1(\text{nlist}[n] \alpha, \alpha) \end{aligned}$$

Both combinators contain internal signals which are invisible to the context as they are “cut off”, from the inside of the relation and out to the surroundings, by the relation π_1 . We have to state an explicit type for those and have chosen to give them the type `nat`. However, we could in fact have given them any non-empty dummy type as it has no impact on the subsequent proofs.

For `rdrf` we prove a type rule and two equality rules expressing how the combinator expands. Some of the rules are shown in Figure 13. The equality rule for the successor case, `rdrf_succ_iff`, is not trivially proven as `rdrfsucc(n)` is not directly defined in terms of `rdrfn` but rather in terms of `colfn`. We use the simplifier to prove the rule, but it turns out that two specific rules is needed (such as `p1_snd_lemma` from Figure 13) to carry through the proof. Therefore we employ our tactic `ProveSimp` which proves the two rules automatically and the proof of `rdrf_succ_iff` is finished by a total of four simplification steps.

For `fork` a similar situation as above arises, since the desired equivalence rule for `forksucc(n)` does not come directly from expanding the definition of `fork`. We first prove that in the row structure, from which the fork is built, the two end

 Figure 12: Graphical interpretation of $\text{rdfr}_3(F)$ and fork_n .

$$\begin{aligned}
 \text{rdfr_type} \quad & \llbracket n \in \text{nat} ; F \in \text{nat} \rightarrow \alpha \times \beta \stackrel{\text{sig}}{\sim} \beta \rrbracket \implies \text{rdfr}_{\beta n}(F) \in \text{nlist}[n] \alpha \times \beta \stackrel{\text{sig}}{\sim} \beta \\
 \text{rdfr_zero_iff} \quad & \text{rdfr}_{\beta 0}(F) = \text{Fst}_{\beta}(\text{NNIL}) ; \text{cross}_{(\text{nlist}[0] \gamma, \beta)} ; \pi_1(\beta, \text{nlist}[0] \text{nat}) \\
 \text{rdfr_succ_iff} \quad & \llbracket n \in \text{nat} ; F \in \text{nat} \rightarrow \alpha \times \beta \stackrel{\text{sig}}{\sim} \beta \rrbracket \implies \\
 & \quad \text{rdfr}_{(\beta, \text{succ}(n))}(F) = \\
 & \quad \text{Fst}_{\beta}(\text{apr}_{\alpha n}^{-1}) ; ((\text{rdfr}_{\beta n}(F) ; \pi_1^{-1}_{(\beta, \text{nlist}[n] \text{nat})}) \downarrow (F^i n ; \pi_1^{-1}_{(\beta, \text{nat})}))) ; \\
 & \quad \pi_1(\beta, \text{nlist}[n] \text{nat} \times \text{nat}) \\
 \text{p1_Snd_lemma} \quad & n \in \text{nat} \implies \text{Snd}_{\beta}(\text{apr}_{\alpha n}) ; \pi_1(\beta, \text{nlist}[\text{succ}(n)] \alpha) = \pi_1(\beta, \text{nlist}[n] \alpha \times \alpha) \\
 \text{fork_succ_iff} \quad & n \in \text{nat} \implies \text{fork}_{(\alpha, \text{succ}(n))} = \text{dub}_{\alpha} ; \text{Fst}_{\alpha}(\text{fork}_{\alpha n}) ; \text{apr}_{\alpha n}
 \end{aligned}$$

Figure 13: rdfr and fork related rules

points will always be equal. This lemma is then used to prove an equivalence rule for $\text{fork}_{\text{succ}(n)}$, expressed in terms of fork_n , as shown in Figure 13.

4.4.2 Proving the Conjectured Rules

The transformation process deriving the convolution circuit made use of the two conjectured rewrite rules, forkmap and hornerf , as shown in Figure 14.

$$\begin{aligned}
 \text{fork_map} \quad & \llbracket n \in \text{nat} ; 0 < n ; R \in \alpha \stackrel{\text{sig}}{\sim} \beta ; \text{function}(R) \rrbracket \implies R ; \text{fork}_{\beta n} = \text{fork}_{\alpha n} ; \text{map}_n(R) \\
 \text{hornerf} \quad & \llbracket R \in \alpha \stackrel{\text{sig}}{\sim} \alpha ; F \in \text{nat} \rightarrow \alpha \times \alpha \stackrel{\text{sig}}{\sim} \alpha ; n \in \text{nat} ; \\
 & \quad \forall n \in \text{nat} \cdot ([R, R] ; F^i n = F^i n ; R) \rrbracket \implies \\
 & \quad [\text{tri}_{\alpha n}(R), \text{pow}_{\alpha n}(R)] ; \text{rdfr}_{\alpha n}(F) = \text{rdfr}_{\alpha n}(\lambda k \in \text{nat} \cdot (\text{Snd}_{\alpha}(R) ; (F^i k))) \\
 \text{pow_comm_F} \quad & \llbracket R \in \alpha \stackrel{\text{sig}}{\sim} \alpha ; F \in \text{nat} \rightarrow (\alpha \times \alpha) \stackrel{\text{sig}}{\sim} \alpha ; n \in \text{nat} ; m \in \text{nat} ; \\
 & \quad \forall n \in \text{nat} \cdot ([R, R] ; F^i n = F^i n ; R) \rrbracket \implies \\
 & \quad [\text{pow}_{\alpha n}(R), \text{pow}_{\alpha n}(R)] ; F^i m = F^i m ; \text{pow}_{\alpha n}(R)
 \end{aligned}$$

Figure 14: Conjectured rewrite rules to be proved

Initially we try to prove forkmap by induction over the size n , but we run into unexpected problems in the base case. By expanding map_0 to NNIL the base case becomes:

$$\llbracket n \in \text{nat} ; R \in \alpha \stackrel{\text{sig}}{\sim} \beta ; \text{function}(R) \rrbracket \implies R ; \text{fork}_{\beta 0} = \text{fork}_{\alpha 0} ; \text{NNIL}$$

The problem is that the right-to-left direction of this equality only holds if R is a non-empty relation. This we cannot conclude since the right side of the equality has no reference to R . An explicit precondition stating that R is non-empty is not desirable in T-Ruby and we therefore instead choose to require that the size parameter n is greater than 0. In this way the base case, now for $n = 1$, refers to R on both sides of the equality and can thus be proved. The key rule in

both the base case and the inductive case is `duplicate` from Figure 7. After a few simplifications the base reaches the form:

$$\llbracket n \in \text{nat} ; R \in \alpha \stackrel{\text{sig}}{\sim} \beta ; \text{function}(R) \rrbracket \implies \text{dub}_\alpha ; [R, R] ; \text{Fst}_\beta(\text{fork}_{\beta 0}) ; \text{apr}_{\beta 0} = \text{dub}_\alpha ; \text{Fst}_\alpha(\text{fork}_{\alpha 0}) ; [\text{NNIL}, R] ; \text{apr}_{\beta 0}$$

which can be solved by `fast_tac.t`. The inductive step can be solved by three simple applications of `Rubysimp` (again several applications is needed to exploit the associativity of serial composition). This is an interesting example of the need for formal proofs. Originally, when entered into T-Ruby, we did not realise that the assumption $n > 0$ was required for `forkmap` to be valid.

The last rule `hornerf` expresses the equivalence of a complex structure to a simpler structure. The pre-condition expresses a distributivity requirement, as in the familiar Horner's rule for algebraic manipulation of polynomials. Such rules are used for many purposes in the Ruby synthesis – in the convolution design, for example, to express timing features, such as the input-output equivalence of a circuit to a systolic version of the same circuit. The graphical interpretation of the rule is given in Figure 15. The convolution example exploits that if R is instantiated to the delay element and F is instantiated to a function to a Pure Ruby relation, then the pre-condition is trivially true due to the retiming theorem.

Figure 15: Graphical interpretation of `hornerf` for $n = 3$

The proof of `hornerf` is performed in RubyZF by induction over the size parameter n followed by a number of rewrite steps. The equality is entered and the induction rule applied:

```
- val prems = goal Convolution.thy
  "!!n. [| R:A<~>A; F:nat ->A*A<~>A; n:nat; ALL n:nat. [[R,R]] ;; F'n = F'n ; R | ] ==>
    [[tri(A,n,R),pow(A,n,R)]] ;; rdrf(A,n,F) =
    rdrf(A,n,lam k:nat. Snd(A,R) ;; F'k)";
- by (eresolve_tac [nat_induct] 1);
Level 1
1. !!n. [| R:A<~>A; F:nat -> A*A<~>A; ALL n:nat. [[R,R]] ;; F'n = F'n ;; R | ] ==>
  [[tri(A,0,R),pow(A,0,R)]] ;; rdrf(A,0,F) =
  rdrf(A,0,lam k:nat. Snd(A,R) ;; F'k)
2. !!n x. [| R:A<~>A; F:nat -> A*A<~>A; ALL n:nat. [[R,R]] ;; F'n = F'n ;; R; x:nat;
  [[tri(A,x,R),pow(A,x,R)]] ;; rdrf(A,x,F) =
  rdrf(A,x,lam k:nat. Snd(A,R) ;; F'k) | ] ==>
  [[tri(A,succ(x),R),pow(A,succ(x),R)]] ;; rdrf(A,succ(x),F) =
  rdrf(A,succ(x),lam k:nat. Snd(A,R) ;; F'k)
```

The base case is solved in one step by Rubysimp using simple equivalences.

```
- by (Rubysimp
  (recrl@[NNIL_NNIL,parcompdist,Fst_def,Id_compl,assoccomp RS sym]) 1);
Level 2
1. !!n x.[| R:A<~>A; F:nat -> A*A<~>A; ALL n:nat.[[R,R]] ;; F'n = F'n ;; R; x:nat;
  [[tri(A, x, R),pow(A, x, R)]] ;; rdrf(A, x, F) =
  rdrf(A, x, lam k:nat. Snd(A, R) ;; F ` k) |] ==>
  [[tri(A, succ(x), R),pow(A, succ(x), R)]] ;; rdrf(A, succ(x), F) =
  rdrf(A, succ(x), lam k:nat. Snd(A, R) ;; F ` k)
```

In the inductive case the recursive combinators are first expanded.

```
- by (Rubysimp (recrl@[assoccomp RS sym,pow_rev,fstpardist]) 1);
Level 3
1. !!n x.[| R:A<~>A; F:nat -> A*A<~>A; ALL n:nat. [[R,R]] ;; F'n = F'n ;; R; x:nat;
  [[tri(A, x, R),pow(A, x, R)]] ;; rdrf(A, x, F) =
  rdrf(A, x, lam k:nat. Snd(A, R) ;; F ` k) |] ==>
  [[apr(A,x)~;[[tri(A,x,R),pow(A,x,R)]];;apr(A,x);apr(A,x)~, R;pow(A,x,R)]];;
  (rdrf(A, x, F) ;; pl(A, nlist[x]nat)~) |] (F ` x ;; pl(A, nat)~) ;;
  pl(A, nlist[x]nat * nat) =
  Fst(A,apr(A,x)~) ;;
  (rdrf(A,x,lam x:nat.Snd(A,R) ;; F`x) ;; pl(A,nlist[x]nat)~) |]
  (Snd(A, R) ;; F ` x ;; pl(A, nat)~) ;; pl(A, nlist[x]nat * nat)
```

A number of manipulations follows, where the most important is the application of par_below.

```
- by (Rubysimp [assoccomp,apr_aprinv,Id_compr] 1);
- by (Rubysimp [fstpardist RS sym, assoccomp] 1);
- by (Rubysimp [par_below,assoccomp RS sym] 1);
- by (Rubysimp [sndpardist RS sym] 1);
Level 7
1. !!n x.[| R:A<~>A; F:nat -> A*A<~>A; ALL n:nat.[[R,R]] ;; F'n = F'n ;; R; x:nat;
  [[tri(A, x, R),pow(A, x, R)]] ;; rdrf(A, x, F) =
  rdrf(A, x, lam k:nat. Snd(A, R) ;; F ` k) |] ==>
  Fst(A,apr(A,x)~) ;;
  (Fst(A,tri(A,x,R)) ;; rdrf(A,x,F) ;; pl(A,nlist[x]nat)~) |]
  (Snd(A,R);[[pow(A,x,R),pow(A,x,R)]];;F`x;;pl(A,nat)~);;
  pl(A,nlist[x]nat*nat) =
  Fst(A, apr(A,x)~);;(rdrf(A,x,lam x:nat.Snd(A,R);;F`x);;pl(A,nlist[x]nat)~) |]
  (Snd(A, R) ;; F ` x ;; pl(A, nat)~) ;; pl(A, nlist[x]nat*nat)
```

We can now exploit the precondition of the rule and apply the lemma pow_comm_F shown in Figure 14. Informally this expresses that if R distributes over $F'n$ then also does $\text{pow}_{am}(R)$. The next large step is the application of the rule belowmid applied in a specific instantiation by inst_rewrite, which also type checks.

```
- by (Rubysimp [pow_comm_F,assoccomp,plinvcomp] 1);
- by (Rubysimp [assoccomp RS sym] 1);
- by (inst_rewrite [("X","pow(A,x,R)"] belowmid [] 1);
- by (Rubysimp [plinvcomp RS sym,assoccomp] 1);
Level 11
1. !!n x.[| R:A<~>A; F:nat -> A*A<~>A; ALL n:nat.[[R,R]] ;; F'n = F'n ;; R; x:nat;
  [[tri(A, x, R),pow(A, x, R)]] ;; rdrf(A, x, F) =
  rdrf(A, x, lam k:nat. Snd(A, R) ;; F ` k) |] ==>
  Fst(A, apr(A, x)~) ;; ((Snd(nlist[x]A, pow(A, x, R)) ;;
  (Fst(A, tri(A, x, R)) ;; (rdrf(A, x, F) ;; pl(A, nlist[x]nat)~))) |]
  (Snd(A, R) ;; (F ` x ;; pl(A, nat)~)) ;; pl(A, nlist[x]nat*nat) =
  Fst(A, apr(A, x)~) ;;
  ((rdrf(A, x, lam x:nat. Snd(A, R) ;; F ` x) ;; pl(A, nlist[x]nat)~) |]
  (Snd(A, R) ;; (F ` x ;; pl(A, nat)~)) ;; pl(A, nlist[x]nat*nat)
```

The last step uses the two rules fstsndcomm and fstsndpar to get the induction hypothesis to match, which finally leads to a completion of the proof.

```
- by (Rubysimp [fstsndcomm RS sym,fstsndpar,assoccomp RS sym] 1);
Level 12
No subgoals!
```

The proof is completed in a total of 12 steps, where the 10 are applications of `Rubysimp` using specific equality rules and the associativity of serial composition in both directions.

4.4.3 Convolution Proof Summary

Above we have defined two new combinators and proved two specific transformation rules used in the convolution example. The complete transformational design of the convolution circuit uses a large number of standard Ruby transformations which all have been proved within RubyZF (rules such as those shown in Section 4.1 and Section 4.2). Finally, the transformation process performed in T-Ruby produces a number of proof obligations mainly originating from the application of conditional transformation rules (e.g. `hornerf`). These proof obligation often express simple facts about basic realations and have all been transferred to RubyZF and proved, for example using the retiming property.

5 Conclusion

We have presented a framework which is well suited to perform proofs of transformation rules generally used in the Ruby design process. A close relationship has been obtained to the existing tool T-Ruby, both in concrete syntax and in the view of Ruby. Both tools are based on the Pure Ruby subset and use this simple basis to construct other circuits and combinators. This should permit an easy connection between the two tools and in this way RubyZF may provide a logical foundation for the T-Ruby system (and other Ruby design platforms), thus ensuring the correctness of a concrete transformation process. RubyZF may easily be used together with other Ruby tools even if they are not based on the Pure Ruby subset, as long as the abstract properties of the combinators are the same. The combinator `mapf` may, for example, be defined inductively instead of in terms of Pure Ruby, but it should still possess the abstract properties shown on page 6.

The development of specialised tactics in connection with type checking considerably increases the productivity and clarity in performing proofs. The user is very seldom bothered with subgoals relating to type checking as this is all done behind the scenes. So even though we have decided to use an expressive, non-decidable type system almost all type goals are solved automatically. Extending `fast_tac` with type checking enabled us to solve a large number of goals automatically which could not be solved with the standard version. Simple rules can be proved fully automatically by `ProveSimp` and even quite complex rules, as for example `hornerf`, are easily proved. The examples in Section 4 clearly show that proofs may be performed at a fairly high level of abstraction and automation using tactics like `fast_tac_t` and `Rubysimp`.

Acknowledgements

Thanks to Robin Sharp for inspiring cooperation with the development of the T-Ruby design system and for showing interest in my project.

References

- [1] A. Dent and K. Hanna. Reasoning about array structures using a dependently typed logic. In *CHDL '93*. Elsevier Science Publishers (North-Holland), 1993.
- [2] Graham Hutton. *Between functions and relations in calculating programs*. PhD thesis, University of Glasgow, 1992.
- [3] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. Elsevier Publishers, 1990.
- [4] G. Jones and M. Sheeran. Relations and refinement in circuit design. In Morgan, editor, *Proc. BCS FACS Workshop on Refinement*. Springer-Verlag Workshop in Computing, 1990.

- [5] L. C. Paulson. Set theory for verification: I. from foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993.
- [6] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.
- [7] O. Rasmussen. A Ruby proof system. Technical Report ID–TR: 1995-161, Dept. of Computer Science, Technical University of Denmark, 1995.
- [8] Lars Rossen. Proving (facts about) ruby. In G. Birtwistle, editor, *The IV Higher Order Workshop, Banff, Workshops in Computing*, pages 265–283. Springer-Verlag, 1990.
- [9] Lars Rossen. Ruby algebra. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits, Oxford 1990, Workshops in Computing*, pages 297–312. Springer-Verlag, 1991.
- [10] R. Sharp and O. Rasmussen. Transformational rewriting with Ruby. In *CHDL '93*, pages 231–248. Elsevier Science Publishers (North-Holland), 1993.
- [11] R. Sharp and O. Rasmussen. The T-Ruby design system. In S. Johnson, editor, *Computer Hardware Description Languages and their Applications, CHDL'95*, pages 587–596. IFIP WG10.5, 1995.
- [12] R. Sharp and O. Rasmussen. Using a language of functions and relations for VLSI specification. In *Functional programming and Computer Architecture, FPCA'95*, pages 45–54, June 1995.