Formalising Ruby in Isabelle ZF

Ole Rasmussen
Dept. of Computer Science
Technical University of Denmark
Email: osr@id.dtu.dk

August, 1995

Abstract

This paper describes a formalisation of the relation based language Ruby in Zermelo-Fraenkel set theory (ZF) using the Isabelle theorem prover. We show how a very small subset of Ruby, called Pure Ruby, easily can be formalised as a conservative extension of ZF and how many useful structures used in connection with VLSI design can be defined from Pure Ruby. The inductive package of Isabelle is used to characterise the Pure Ruby subset by an inductive definition, to allow proofs to be performed by structural induction over the Pure Ruby elements. Finally we demonstrate how various kinds of proofs may be automated and the explicit type checking of ZF hidden by the definition of specialised tactics.

1 Introduction

Ruby [2] is a relation based language intended for specifying VLSI circuits. A circuit is described by a binary relation between appropriate, possibly complex domains of values, and simple relations can be combined into more complex relations by a variety of combining forms. The Ruby relations generate an algebra which defines a set of equivalences. These equivalences are used in the Ruby design process which typically involves a transformation from a "specification" to an "implementation" both expressed in Ruby. The implementation describes the same (or a smaller) relation as the specification but in another form, which in a given interpretation is said to be implementable. This design style is referred to as *design by calculation* and is demonstrated in [3, 7, 9].

To support this style of design we have constructed a tool called T-Ruby, which is based on a formalisation of Ruby as a language of functions and relations [10]. The T-Ruby system enables the user to perform the desired transformations in the course of a design, to simulate the behaviour of a class of implementable relations, and to translate the final Ruby description of such relations into a VHDL description for subsequent synthesis by a high-level synthesis tool.

This paper describes a formalisation of Ruby, called RubyZF, within the Isabelle theorem prover [6] using a formulation of Zermelo-Fraenkel set theory (ZF). The work follows the line started by Rossen [8] and continued in the T-Ruby system. 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. by proofs of various refinement steps.

ZF was a natural choice for Ruby since the basic objects of Ruby are relations, which are conventionally modelled as sets of pairs. The basic objects of ZF are in fact sets in contrast to for example HOL [1] where the basic objects are functions. Set theory has a tremendous expressive power and its few basic concepts are well understood. Usually it is regarded as clumsy and not very well suited for doing automated proofs but with the extensive work of e.g. Larry Paulson, it has become possible to use ZF. This means that from a Ruby point of view ZF is natural and from a ZF point of view Ruby is feasible.

Several factors favoured the use of the Isabelle theorem prover. Naturally having the development of ZF in the standard distribution of Isabelle meant that the formalisation of Ruby could start at a reasonable high level. Secondly, the fairly high degree of automation available in Isabelle was interesting as many of the proofs in Ruby follow the same pattern (equality proofs). Finally Isabelle's advanced parsing and pretty printing features were important since RubyZF is meant to be used directly in connection with T-Ruby. The user should not be bothered with too may syntax variations.

2 Introduction to Ruby

The definition of Ruby used in this work is based on the so-called Pure Ruby subset as introduced by Rossen [8]. This makes use of the observation that a very large class of the relations which are useful for describing VLSI circuits can 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 as shown in Figure 1. These four are binary relations and the notation aRb means that a is related to b by R, and is synonymous with $(a, b) \in R$.

$$\begin{aligned} a\ (\mathsf{spread}(f))\ b &\triangleq \forall t \in \mathbb{Z} \cdot (a(t)\ f\ b(t)) & (1)\\ a\ \mathcal{D}\ b &\triangleq \forall t \in \mathbb{Z} \cdot a(t) = b(t+1) & (2)\\ a\ (F\ ;G)\ b &\triangleq \exists c \cdot (a\,F\,c\ \wedge\ c\,G\,b) & (3)\\ \langle a_1, a_2\rangle\ [F, G]\ \langle b_1, b_2\rangle &\triangleq a_1\,F\,b_1\ \wedge\ a_2\,G\,b_2 & (4) \end{aligned}$$

Figure 1: The basic elements of Pure Ruby

In the figure the variables $a, b \ldots$ are of type $\mathsf{sig}(\mathcal{T})$, where $\mathsf{sig}(\mathcal{T})$ is the type of streams of values of type $\mathcal{T}$. This is usually represented as a function of type $\mathbb{Z} \to \mathcal{T}$, where we identify $\mathbb{Z}$ with 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. Thus signals can be expressed as:

$$\begin{aligned} \mathsf{sig} &= \mathsf{time} \to \mathsf{ChTy}\\ \mathsf{ChTy} &= \mathsf{BasChTy}\\ &\mid\ \mathsf{ChTy} \times \mathsf{ChTy}\\ &\mid\ \mathsf{nlist}[n](\mathsf{ChTy}) \end{aligned}$$

where $\mathsf{nlist}[n]\alpha$ are lists of length n. Since nlists are parameterised in n, Ruby relations may have dependent product types. The base types, $\mathsf{BasChTy}$, will typically be natural numbers, bits etc. but no explicit restriction is made. Note that Ruby relations are binary relations on single signals such that composite signals are always represented as a function from time to the composite type. Thus in equation 4 above $\langle a_1, a_2\rangle$ does not stand for a conventional ordered pair of two signals but rather for the paring of two signals into one signal.

Viewed as relations $\mathsf{spread}(f)$ is the lifting to streams of the pointwise relation described by f. If f is a relation of type $\alpha \sim \beta$ (the type of binary relations between values of type α and type β) then $\mathsf{spread}(f)$ is of type $\mathsf{sig}(\alpha) \sim \mathsf{sig}(\beta)$. For notational convenience, and to stress the idea that it describes the lifting to streams of a pointwise relation of type $\alpha \sim \beta$, this type will be denoted $\alpha \overset{sig}{\sim} \beta$. $\mathcal{D}$ – the delay element – relates a stream to another stream which has an offset of one time tick, $(F\ ;G)$ describes relational composition and $[F,G]$ relational product.

If we view the four Pure Ruby elements as circuits then $\mathsf{spread}(f)$ describes the synchronously clocked combinational circuit with the functionality of f. $\mathcal{D}$ describes the basic sequential circuit (a latch), $(F\ ;G)$ sequential composition of two circuits and $[F,G]$ parallel composition.

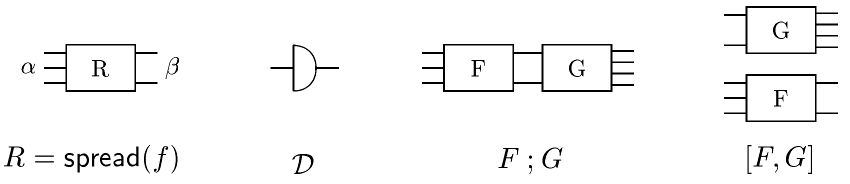

Figure 2: Graphical interpretations of the four Pure Ruby elements

A feature of Ruby is that relations and combinators not only have an interpretation in terms of circuit elements, but also have a natural *graphical interpretation*, corresponding to an abstract floorplan for the circuits which they describe. The conventional graphical interpretation of spread (or, in fact, of any other circuit whose internal details we do not wish to show) is as a labelled rectangular box. The components of the domain and range are drawn as wire stubs, whose number reflects the types of the relations in an obvious manner: a simple type gives a single stub, a pair type two and so on. 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 2.

3 Introduction of Isabelle/ZF and RubyZF

The purpose of this section is to give a brief introduction to the Isabelle logical framework and the concrete logic used in this work. A thorough presentation of the system is given in [6]. Isabelle is a generic logical framework meant for defining different proof systems. 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 the simple typed lambda calculus. Inference rules and axioms in Isabelle are all theorems of the meta logic, usually containing a meta implication.

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 high-order SML functions. The simple tacticals are used for composing tactics sequentially, alternatively, repeatedly, but also more complex tacticals exist for expressing control structures, depth-first search etc.

Isabelle has a built in generic parsing mechanism allowing a flexible mixfix notation of new defined symbols in the object logic. Furthermore the user may define *translations*, which will only affect the appearance of an object, keeping the internal representation unchanged.

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 [4]. In ZF desired typing of variables is stated directly in the goal assumptions. The Isabelle meta type system is used very little in ZF since only two types are defined: `i` for sets and `o` for propositions. Figure 3 shows the correspondence of mathematical and ASCII notation of a selection of symbols used in this work (a mixture of meta and ZF symbols). Furthermore the notation `[|P1;...;Pn|]==>Q` is a shorthand for nested implication.

$a \in A$	`a:A`
$A \subseteq B$	`A<=B`
(a, b)	`<a,b>`
$A \times B$	`A * B`
$A \to B$	`A -> B`

$\forall x \cdot P(x)$	`ALL x.P(x), !!x.P`
$\exists x \cdot P(x)$	`EX x.P(x)`
$P \Rightarrow Q$	`P ==> Q`
$\lambda_{x \in A} \cdot b(x)$	`lam x:A.b(x), %x.b`
$f(a)$	``f`a, f(a)``

Figure 3: ASCII notation of selected symbols

3.1 Generic Packages

Three generic packages have been used extensively in this work and are all set up in the standard distribution of ZF; the *simplifier*, the *classical* reasoning package and the *inductive* definition package.

The simplifier performs conditional and unconditional rewriting using contextual information and takes a simplification set as argument. The set contains a number of rewrite rules and possibly the specification of tactics to be applied in specific cases. The simplifier can both rewrite the assumptions and the conclusion.

The classical reasoning package provides a number of tactics to prove theorems in the style of the sequent calculus. They take a classical rule set as argument, which contains a collection of introduction and elimination rules divided into either safe and unsafe rules. Safe rules can be applied blindly but unsafe rules must be handled more carefully. For example `fast_tac` tries to solve a goal completely using depth-first search.

The inductive package is based on a fixedpoint approach and permits the formalisation of all monotone inductive definitions [5]. The inductive definitions are expressed as a subset of an existing set given the desired inference rules as Isabelle meta implications. The package is used in Section 4.4.

3.2 Customising ZF to Ruby Proofs

Setting up rule lists of related rules to a specific domain speeds up the process of developing proofs considerably. Furthermore it is a great help for the user to be able to lift the reasoning to a more general level. In RubyZF three main lists of rules are defined: `Ruby_type`, `RubyI` and `RubyE` containing Ruby type rules, introduction rules and elimination rules respectively. Additionally the type list includes type theorems for nlist-constructors, signal-constructors and arithmetic operations, as Ruby proofs generally involve reasoning about these. Finally a classical rule set, `RubyZF_cs`, is set up as an extension to the rule set `ZF_cs` provided in Isabelle. The rules from `RubyI` are added to `ZF_cs` as introduction rules and the rules from `RubyE` are added as *safe* elimination rules.

3.2.1 Type Checking Tactics

ZF is basically untyped, but types can of course be modelled as sets of values. This means there is no distinction between type-goals and other goals. However, type checking is tedious and would take up a large amount of time working in ZF. Therefore a number of specialised tactics have been developed in RubyZF to solve type goals automatically. If normal resolution is used, the type checking tactics may instantiate variables inappropriately, thus leading to unsolvable proofs. Type checking therefore makes use of the tactic `typechk_tac` (included in ZF) which solves any subgoal in the complete proof state of the form $a \in A$, where a is not a schematic variable. To be able to update the type information interactively and enable the tactics to use the updated information, all the rules from `Ruby_type` are also stored in an SML reference variable called `basictypeinfo`. The following type checking tactics are provided:

`typeit` *trls* uses `typechk_tac` with the supplied theorems *trls* plus the theorems contained in `basictypeinfo`.

`typchk` *trls tac* first applies *tac* and then type checks the proof state as `typeit`.

`resolve_tac_c` *trls thms i* performs a '`resolve_tac` *thms i*' and then type checks the proof state as above.

`eresolve_tac_c` *trls thms i* performs a '`eresolve_tac` *thms i*' and then type checks the proof state as above.

Typically the tactics are applied with no extra type information (*trls* is empty) since all necessary type information is contained in `basictypeinfo`.

Special versions of the classical tactics are provided to make these tactics more useful in proofs with many type conditions. The original names of the tactics are kept but now suffixed with an '`_t`' and take an extra argument containing the extra type rules to be used (apart from the rules in `basictypeinfo`). For example the typed version of `fast_tac` is called `fast_tac_t`.

4 Formalising Pure Ruby

This section will take us through the formalisation of Ruby in standard Zermelo-Fraenkel set theory implemented in Isabelle. We describe a semantical embedding of the Ruby algebra in ZF, i.e. we define new Ruby constructors directly as a conservative extension of ZF. We formalise the basic concepts of Pure Ruby and first define a theory of signals, then the four Pure Ruby elements and finally construct a set containing all Pure Ruby relations of certain types.

Much work has been put into hiding the implementation and most of the set-theoretical reasoning, for example by exploiting the advanced parsing and pretty-printing features in Isabelle. We follow the same style of development as described by Paulson [4] and introduce standard introduction and elimination rules for all new constructors and a large number of rewrite rules to lift the reasoning close to the Ruby-level.

4.1 Signals

The theory of signals defines the set of streams and a number of functions to combine and destruct signals. The theory is based on a theory defining the fixed length lists, nlists, which is not presented in this paper. However, nlists are defined in terms of `listn` from the ZF distribution together with the usual operations on nlists such as: nnil, the nil-element; ncons, to concatenate an element to the front of the list; and nsnoc, to concatenate an element to the back of the list. The need for nsnoc will be apparent in connection with the recursive combinators in Section 5.3. The definition of the theory `signal` is shown in Figure 4.

```
Signal = Nlist + Univ +
consts
  time,ChTy,BasChTy,ChEl,snil :: "i"
  sig,pri,sec                 :: "i => i"
  spair                       :: "[i,i]=>i"   ("(<_#/_>)")
  scons                       :: "[i,i,i]=>i" ("([_@>_|/_])")
  ssnoc                       :: "[i,i,i]=>i"   ("([_<@_|/_])")

translations
  "sig(A)"      ==      "time -> A"

defs
  time_def  "time     == integ"
  snil_def  "snil     == (lam t:time.nnil)"
  spair_def "<a # b>  == (lam t:time.<a`t,b`t>)"
  scons_def "[a@>l|n] == (lam t:time.ncons(n,a`t,l`t))"
  ssnoc_def "[l<@a|n] == (lam t:time.nsnoc(n,l`t,a`t))"

  pri_def   "pri(a)   == (lam t:time.fst(a`t) )"
  sec_def   "sec(a)   == (lam t:time.snd(a`t) )"

  chel_def  "ChEl     == univ(Union(BasChTy))"
  chty_def  "ChTy     == Pow(ChEl)"
rules
  nat_in_base    "nat:BasChTy"
end
```

Figure 4: Definition of signals

Signals make use of the Isabelle facility to define translations and are represented as functions from time to a type, where time is integers (from the standard distribution). Four constructor functions are defined for signals: `snil` constructs a signal from `nnil`, `spair` pairs two signals and `scons` and `ssnoc` concatenates a signal to a signal list from the front or the back respectively. Type rules for each of them are proved automatically by the tactic `typeit`:

```
snil_type     snil:sig(nlist[0]A)
spair_type    [|a:sig(A);b:sig(B)|] ==> <a#b>:sig(A*B)
scons_type    [|a:sig(A);l:sig(nlist[n]A)|]==> [a@>l|n]:sig(nlist[succ(n)]A)
ssnoc_type    [|a:sig(A);l:sig(nlist[n]A)|]==> [l<@a|n]:sig(nlist[succ(n)]A)
```

In the subsequent Ruby proofs we need elimination rules for four different cases of signals corresponding to the constructors above. The elimination rule for signal pairs is proved for example by exploiting extensionality of functions and pairs. The other three are proved in a similar fashion:

```
sig_pairE     [| c:sig(A*B);
                 !!a b.[| c=<a#b>;a:sig(A);b:sig(B) |]==>P |]==> P
sig_nlist0E   [| c:sig(nlist[0]A); c=snil==>P |]==> P
sig_nlistE    [| c:sig(nlist[succ(n)]A);
                 !!a l.[| c=[a@>l|n];a:sig(A);l:sig(nlist[n]A)|]==>P|]==>P
sig_ssnocE    [| c:sig(nlist[succ(n)]A);
                 !!a l.[| c=[l<@a|n];a:sig(A);l:sig(nlist[n]A)|]==>P|]==>P
```

Two destructor functions, `pri` and `sec`, are defined for signal pairs and the expected equality rules proved. Furthermore signal pairing and concatenation have the conventional injection properties:

```
pri_iff       a:sig(A) ==> pri(<a#b>) = a
sec_iff       b:sig(B) ==> sec(<a#b>) = b

spair_inject  [| <x1#x2> = <x1a#x2a>; x1:sig(A); x2:sig(B);
                  x1a:sig(C); x2a:sig(D);
                  [| x1 = x1a; x2 = x2a |] ==> P |] ==> P
scons_inject  [| [a@>la|n] = [b@>lb|n]; a:sig(A); b:sig(B);
                  la:sig(nlist[n]C); lb:sig(nlist[n]E);
                  [| la = lb; a = b |] ==> P |] ==> P
scons_iff     [| a:sig(A); b:sig(B); la:sig(nlist[n]C);
                  lb:sig(nlist[n]E) |] ==>
                  [a@>la|n] = [b@>lb|n] <-> la = lb & a = b
```

The lower part of Figure 4 defines the sets of channel elements and channel types. In Section 4.4 we will need a set which is large enough to contain all Ruby channel types, in order to define the set of Pure Ruby relations inductively. The two following theorems state that both pairs and nlists are contained in the channel types:

```
prod_in_chty  [| A:ChTy; B:ChTy |] ==>       A*B: ChTy
nlist_in_chty [| A:ChTy; n:nat  |] ==> nlist[n]A: ChTy
```

Finally the defined rule `nat_in_base` is an example of how new types can be added to the set of base channel types.

4.2 Relational definitions

This theory defines various sets describing the values of the domain and range of signal relations. The definitions are given in Figure 5 together with syntactic translations for relations, `~`, and signal relations, `<~>`.

The set `dtyp(R)` defines a subset of A if $R \in A \overset{sig}{\sim} B$. These sets are needed in subsequent

```
Relation = Signal +
consts
  dtyp,rtyp,ddtyp,rdtyp,
  rrtyp,drtyp  ::   "i=>i"
  "@rel"       ::   "[i,i]=>i"   ("(_~/_)"   [73,72] 72)
  "@srel"      ::   "[i,i]=>i"   ("(_<~>/_)" [73,72] 72)

translations
  "A~B"     ==  "Pow(A * B)"
  "A<~>B"   ==  "sig(A)~sig(B)"

defs
  dtyp_def    "dtyp(R)    == range(Union(domain(R)))"
  rtyp_def    "rtyp(R)    == range(Union(range(R)))"
  ddtyp_def   "ddtyp(R)   == domain(dtyp(R))"
  rdtyp_def   "rdtyp(R)   == range(dtyp(R))"
  rrtyp_def   "rrtyp(R)   == range(rtyp(R))"
  drtyp_def   "drtyp(R)   == domain(rtyp(R))"
end
```

Figure 5: Auxiliary definitions on relations

proofs in connection with simple combinators. For `dtyp` the following properties can be proved:

```
dtyp_sig       [| <x,y>:R; x:sig(A) |] ==> x:sig(dtyp(R))
dtyp_rel       [| R <= sig(A)*sig(B); x:sig(dtyp(R)) |] ==> x:sig(A)
sub_dtyp_rtyp  R:A<~>B==> R:dtyp(R)<~>rtyp(R)
```

Similar properties are proved for the four sets defined for relations on signal pairs (relations for which $R \in A \times B \overset{sig}{\sim} C \times D$), i.e. the sets `ddtyp` etc.

4.3 Pure Ruby

The four Pure Ruby elements introduced in Section 2 are defined in ZF as shown in Figure 6. ZF does not have a special notion of types, as for example HOL does, so all types must be supplied explicitly. However for `spread`, serial and parallel composition the types can be inferred from the relational arguments. This unfortunately is not possible in the case of the delay element, thus the type must be given as a parameter. In contrast to conventional Ruby notation serial composition is written as two semi-colons to distinguish it from Isabelle's notation for nested implication, and parallel composition is written with double square brackets to distinguish it from ZF list notation.

For each of the four elements of Pure Ruby, type, introduction and elimination rules are proved. For the `spread` element the rules are:

```
spread_type f:A~B ==> spread(f):A<~>B
spreadI     [| ALL t:time.<x`t,y`t>:f; f:A~B; x:sig(A); y:sig(B) |] ==> <x,y>:spread(f)
spreadE     [| <x,y>:spread(f); [|ALL t:time.<x`t,y`t>:f |]==> P |] ==> P
```

Similar rules are proved for the delay element ($\mathcal{D}$). Also the rules for serial and parallel composition are very similar. The rules for parallel composition reflect the fact that it relates

```
PureRuby  = Relation +
consts
  spread,D    ::   "i=>i"
  par         ::   "[i,i]=>i" ("([[_,/_]])" )
  "@ser"      ::   "[i,i]=>i" ("(_ ;;/ _)" [60,61] 60 )

defs
  spread_def "spread(f)== {xy:sig(domain(f)) * sig(range(f)).
                          EX x y. xy=<x,y> & (ALL t:time.<x`t,y`t>:f)}"
  delay_def  "D(A)     == {xy:sig(A) * sig(A).
                          EX x y. xy=<x,y> &
                             (ALL t:time.x`t = y`(t $+ $#1))}"
  comp_def   "R ;; S   == {xz:domain(R) * range(S).
                          EX x z y. xz = <x,z> & <x,y>:R & <y,z>:S}"
  par_def    "[[R,S]]  == {xy:(sig(dtyp(R) * dtyp(S))  * sig(rtyp(R) * rtyp(S))).
                          EX x y. xy = <x,y> &
                             <pri(x),pri(y)>:R & <sec(x),sec(y)>:S }"
end
```

Figure 6: Definition of the four Pure Ruby elements

signal pairs and not conventional pairs:

```
par_type  [| R:A<~>B;  S:C<~>E |] ==> [[R,S]]:(A*C)<~>(B*E)
parI      [| <x1,y1>:R; <x2,y2>:S; x1:sig(A); x2:sig(B);
             y1:sig(C); y2:sig(E) |] ==> <<x1#x2>,<y1#y2>>:[[R,S]]
parE      [| <<x1#x2>,<y1#y2>> :[[R,S]]; [| <x1,y1>:R; <x2,y2>:S |]==> P;
            x1:sig(A); x2:sig(B); y1:sig(C); y2:sig(E) |] ==> P
```

To verify that the above definitions in ZF actually describe the same Pure Ruby constructors as defined in Section 2, we prove that they enjoy the mathematical properties expected. As an example let us look at the equality rule for serial composition:

```
comp_iff  [| R:A<~>B; S:B<~>C; x:sig(A); z:sig(C) |] ==>
            <x,z>: R;;S  <-> (EX y:sig(B). <x,y>:R & <y,z>:S)
```

The theorem is proved in one step by `fast_tac`. It is easily seen that this theorem expresses the same property as the definition of serial composition given in Section 2 (Equation 3) under the assumption that the types are correct.

4.3.1 Distributivity of Serial and Parallel Composition

A useful example of the use of the typed tactics is to prove equality rules in the Ruby algebra. Here we prove the distributivity of serial and parallel composition which can be expressed as:

$$\forall_{R\,\in\,\alpha_1 \overset{sig}{\sim} \beta_1}\forall_{S\,\in\,\alpha_2 \overset{sig}{\sim} \beta_2}\forall_{T\,\in\,\beta_1 \overset{sig}{\sim} \gamma_1}\forall_{U\,\in\,\beta_2 \overset{sig}{\sim} \gamma_2} \cdot [R,S]\,;[T,W] = [(R\,;T),(S\,;W)]$$

This is entered into Isabelle as a meta implication with the relation types as premises.

```
- val prems = goal PureRuby.thy
    "[| R:A1<~>B1; S:A2<~>B2; T:B1<~>C1; W:B2<~>C2 |] ==>
                [[R,S]] ;; [[T,W]] = [[(R ;; T),(S ;; W)]]";
Level 0
 1. [[R,S]] ;; [[T,W]] = [[R ;; T,S ;; W]]
```

Many proofs in connection with Ruby consist of proving equalities of relational expressions written in a pointfree notation. Therefore a special tactic, `prove_equal`, has been developed to start such proofs. An equality is split into two implications and appropriate data values are added to the relations and typed properly according to the premises.

```
- by (prove_equal prems 1);
Level 1
 1. !!x y.[| <x, y> : [[R,S]] ;; [[T,W]]; x : sig(A1 * A2);
         y : sig(C1 * C2) |] ==>  <x, y> : [[R ;; T,S ;; W]]
 2. !!x y.[| <x, y> : [[R ;; T,S ;; W]]; x : sig(A1 * A2);
         y : sig(C1 * C2) |] ==>  <x, y> : [[R,S]] ;; [[T,W]]
```

Both these subgoals can be solved by the typed version of the classical prover, `fast_tac_t`, using the classical rule-set `RubyZF_cs`:

```
- by (ALLGOALS (fast_tac_t RubyZF_cs prems));
No subgoals!
```

The goals cannot be solved by the standard version of `fast_tac` as it will make wrong instantiations of type variables.

4.4 The Ruby Relation Type

To be able to reason about Ruby relations by structural induction over the four elements of Pure Ruby, we construct a set containing all Pure Ruby relations. We define the set `pure` inductively, as shown in Figure 7, using Isabelle's inductive package. Each element of `pure` is a triple containing a relation, its domain type and its range type which are subsets of signal relations on channel elements, on channel types and channel types respectively (the `domains` field). The inductive set is characterised by four introduction rules, one for each Pure Ruby element, as stated in the `intrs` field. Three specialised type rules are needed to typecheck the inductive definition:

```
spread_chel_rel [| A:ChTy; B:ChTy; f:A~B |] ==> spread(f):ChEl<~>ChEl
D_chel_rel      A:ChTy ==> D(A):ChEl<~>ChEl
par_chel_rel    [| S:ChEl<~>ChEl; R:ChEl<~>ChEl|]==> [[R,S]]:ChEl<~>ChEl
```

The above type rules together with a few others are stated in the `type_intrs` field. The inductive definition then returns a number of theorems e.g. the four introduction rules, an elimination rule and an induction rule.

The set of Pure Ruby relations, `<R>`, is defined as the subset of signal relations which belong to the set `pure`. Pure Ruby type rules are proved in one step by `fast_tac_t` using the introduction rules from above inductive definition:

```
spreadR         [| A:ChTy; B:ChTy; f:A~B |] ==>spread(f):A<R>B
delayR          A:ChTy ==> D(A):A<R>A
compR           [| R:A<R>C; S:C<R>B      |] ==>    R;;S : A<R>B
parR            [| R:A1<R>B1; S:A2<R>B2  |] ==> [[R,S]] : A1*A2<R>B1*B2
```

The rule expressing structural induction over Pure Ruby relations is easily proved by the induction theorem for `pure`. Note that the predicate `P` is a function of both the relation and

```
RubyType = PureRuby +
consts
  pure        ::  "i"
  "@rrel"     ::  "[i,i]=>i"      ("(_<R>/_)" [73,72] 72)

inductive
  domains    "pure" <= "(ChEl<~>ChEl)*ChTy*ChTy"
  intrs
    spread   "[| A:ChTy; B:ChTy; f:A~B |] ==>   <spread(f),A,B>:pure"
    delay    "A:ChTy ==> <D(A),A,A>: pure"
    comp     "[| <R,A,B1>: pure; <S,B2,C>: pure|]==> <R;;S,A,C>:pure"
    par      "[| <R,A1,B1>:pure; <S,A2,B2>:pure|]==>
                                     <[[R,S]],A1*A2,B1*B2>:pure"
  type_intrs "[spread_chel_rel,D_chel_rel,comp_type,
               par_chel_rel,prod_in_chty]"
defs
  rrel_def    "A<R>B== {r:A<~>B. <r,A,B>:pure}"
end
```

Figure 7: Definition of the Pure Ruby set

its type:

```
rubyrel_induct  [|R: A<R>B ;
                  !!f A B.f:A~B ==> P(A,B,spread(f));
                  !!A.P(A,A,D(A));
                  !!R S A B1 B2 C.[| R:A<R>B1; S:B2<R>C ;P(A,B1,R);
                                   P(B2,C,S)|] ==> P(A,C,R;;S) ;
                  !!R S A1 A2 B1 B2.[| R:A1<R>B1; S:A2<R>B2; P(A1,B1,R);
                                   P(A2,B2,S)|] ==> P(A1*A2,B1*B2,[[R,S]])
                |]  ==> P(A,B,R);
```

5 Circuits and Combinators

This section uses the formalisation of Pure Ruby from the previous section as a platform for defining various Ruby *circuits* and *combinators* commonly used in connection with VLSI design. Circuits are generally non-parameterised signal relations and the combinators are parameterised signal relations typically combining signal relations into new signal relations. First we introduce a suite of relations, usually known as *wiring relations*, describing different kinds of wiring patterns as lifted pointwise relations. Many of these wiring relations are used to define a number of combining forms which combine one or more relations into new relations (similar to what we have seen for serial and parallel composition). The circuits and combinators are defined solely in terms of Pure Ruby and the combinators are divided into either simple or recursive ones.

For each circuit and combinator the same 4 kinds of rules are proved: a signal type rule, a Ruby type rule, an introduction rule and an elimination rule. The two type rules state the general signal relation type and that the relation belongs to the set of Pure Ruby relations. The introduction and elimination rules follow the same pattern as we saw in Section 4.3 for the Pure Ruby elements.

5.1 Wiring Relations

In Ruby various types of component inter-connections are described by lifted pointwise relations and are often known as wiring relations. They are typically used to "glue together" adjacent relations by getting the respective wiring patterns to match. In this section we present a number of the basic wiring relations together with a suite of wiring relations used in connection with nlists and thus parameterised in their size.

The relation ι (Id(A)) is the polymorphic identity relation for all types A and relates two equal signals of data. reorg describes the conversion between the two possibilities of pairing three elements. cross relates a pair of values to the reversed pair. Finally dub relates a stream to two copies of the same stream thus describing a conventional fork. The standard graphical interpretations of some of the wiring relations, together with their (polymorphic) types of the signals, are shown in Figure 8 and clearly expresses their intended behaviour.

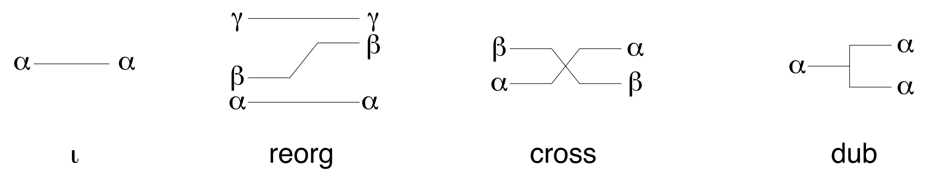

Figure 8: Graphical interpretation of simple wiring relations

All definitions of wiring relations follow the same pattern and are the lifting of a combinational relation by the spread element. Unlike conventional Ruby notation all wiring relations have to be explicitly parameterised in their types. The definitions are shown in Figure 9. The two relations lwir and rwir describes special wiring patterns which will be used in the next section to define relational inverse.

We show a signal type rule, a Ruby type rule, an introduction rule and an elimination rule for the wiring relation reorg:

```
reorg_type reorg(A,B,C): (A*B)*C<~>A*B*C
reorgR     [|A:ChTy;B:ChTy;C:ChTy|] ==> reorg(A,B,C): (A*B)*C<R>A*B*C
reorgI     [|a:sig(A);b:sig(B);c:sig(C) |] ==>
                     < <<a#b>#c>,<a#<b#c>> >:reorg(A,B,C)
reorgE     [| < <<a#b>#c>,<d#<e#f>> >:reorg(A,B,C) ;[|a=d;b=e;c=f|]==> P;
              a:sig(A');b:sig(B');c:sig(C');d:sig(A'');e:sig(B'');
              f:sig(C'')|] ==>  P
```

Recursive combinators usually relates signals of nlists as will be seen in Section 5.3 and we therefore need to be able to concatenate single elements to either side of the nlists. Since all definitions in Ruby are given in a pointfree notation these operations are also defined as relations on signals.

The relation apl_n (append left) relates an element a_1 and a list of n elements a_2 to a list of $n+1$ elements where a_1 is concatenated to the front of a_2. The relation apr_n (append right) is similar to apl_n but the element is appended to the back of the list. The graphical interpretations of apl_n and apr_n are depicted for a specific size, n, in Figure 10.

The definitions of the nlist related wiring relations are shown in the lower part of Figure 9. The relation NNIL relates two empty signal nlists of any type. As for reorg above four rules

```
Wiring = RubyType +
consts
  Id,dub                         :: "i=>i"
  NNIL,cross,lwir,rwir,apl,apr :: "[i,i]=>i"
  reorg                          :: "[i,i,i] => i"

defs
  Id_def    "Id(A)       == spread({xy:A*A.EX x. xy=<x,x>})"
  lwir_def  "lwir(A,B)   == spread({ab:A*(A*(B*B)).
                              EX a b.ab = <a,<a,<b,b>>>})"
  rwir_def  "rwir(A,B)   == spread({ab:(A*(A*B)) *B .
                              EX a b.ab = <<a,<a,b>>,b>})"
  reorg_def "reorg(A,B,C)== spread({ab: ((A*B)*C)*(A*(B*C)).
                              EX a b c.ab =<<<a,b>,c>,<a,<b,c>>>})"
  cross_def "cross(A,B)  == spread({ab: ((A*B)*(B*A)).
                              EX a b. ab = <<a,b>,<b,a>>})"
  dub_def   "dub(A)      == spread({ab: (A*(A*A)). EX a. ab = <a,<a,a>>})"
  NNIL_def  "NNIL(A,B)   == spread({ab:(nlist[0]A * nlist[0]B).
                              ab = <nnil,nnil>})"
  apl_def   "apl(A,n)    == spread({ab:(A*nlist[n]A) * nlist[succ(n)]A.
                              EX a1 a2.ab = <<a1,a2>,ncons(n,a1,a2)>})"
  apr_def   "apr(A,n)    == spread({ab:(nlist[n]A*A) * nlist[succ(n)]A.
                              EX a1 a2.ab = <<a1,a2>,nsnoc(n,a1,a2)>})"
end
```

Figure 9: Definition of simple wiring relations

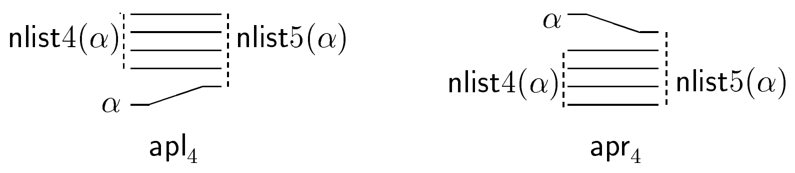

Figure 10: Graphical interpretation of apl$_n$ and apr$_n$ for $n = 4$.

are proved as e.g. for apr:

```
apr_type  apr(A,n):nlist[n]A * A<~>nlist[succ(n)]A
aprR      A:ChTy ==> apr(A,n):nlist[n]A * A<R>nlist[succ(n)]A
aprI      [|a:sig(A); l:sig(nlist[n]A)|] ==> <<l#a>,[l<@a|n]>:apr(A,n)
aprE      [| <<la#a>,[lb<@b|n]>:apr(A,n);a:sig(A1);b:sig(A2);
             la:sig(nlist[n]A3); lb:sig(nlist[n]A4);
             [|a=b;la=lb |] ==> P  |] ==>  P
```

5.2 Simple Combinators

Combinators are usually high-order functions which, given appropriate relational arguments, return signal relations. All the definitions are given in a pointfree notation in terms of Pure Ruby elements. This has the advantage of maintaining the simplicity of the basic theory, of permitting structural induction, and general transformation rules. Unfortunately it also makes the definitions look very complicated and we therefore show that they are equivalent to the more conventional definitions using data values. Thus for all combinators we prove the conventional type, introduction and elimination rules plus the above equality rule.

Figure 11 demonstrates how relational inverse can be defined in terms of Pure Ruby using the wiring relations lwir and rwir. Furthermore the figure illustrates the graphical interpretation of Fst and Snd.

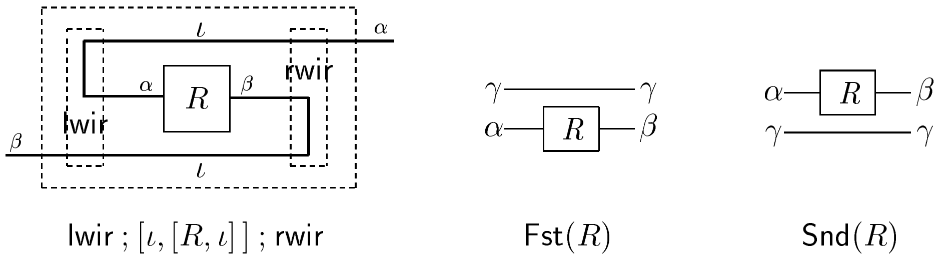

Figure 11: Graphical interpretation of inversion, Fst and Snd

In Figure 12 the definitions of some of the simple combinators are given. As mentioned in Section 4.3 all type arguments must be given explicitly in ZF, but in many cases they can be deduced from the relational arguments by dtyp, rtyp etc.

We first prove that the non-parameterised version of inversion, ~, equals the type parameterised version, inv'. I.e. that no information is lost when deducing the types from the relations:

```
inv'_iff    R:A<~>B ==> R~ = inv'(A,B,R)
```

Using the above theorem the 4 conventional rules are easily proved. Clearly inverse of R is a Pure Ruby relation if R is (stated by invR) as it is only constructed from Pure Ruby elements. The last theorem, inv_iff, shows that the definition actually corresponds to the more conventional definition using data values.

```
inv_type    R:A<~>B ==> R~ :B<~>A
invR        R:A<R>B ==> R~ :B<R>A
invI        [| <b,a>:R; a:sig(A); b:sig(B) |] ==> <a,b>:R~
invE        [| <a,b>: R~;R:A<~>B; [| <b,a>:R |] ==> P |] ==>  P
inv_iff     R:A<~>B ==> <a,b>: (R~) <-> <b,a>:R
```

The combinator $\updownarrow$ (called *below*, and written as || in Isabelle) describes partial composition for two relations whose domain and range types are both pair types: in $R \updownarrow S$, the second component in the domain of R is connected to the first component of the range of S. The graphical interpretation of this is with R *below* S as shown in Figure 13. The dashed lines to the left suggest how *below* is constructed from Pure Ruby.

The Isabelle definition of below is given in the lower part of Figure 12. We first prove that the non-parameterised version, ||, equals the type parameterised version, below':

```
below'_iff    [|R :A*B<~>C*De;  S:E*F<~>B*G |] ==>
                   R || S = below'(A,B,C,De,E,F,G,R,S)
```

We prove a similar family of rules to the above, where the last rule again expresses that

```
SimpComb = Wiring +
consts
  inv'          ::      "[i,i,i]=>i"
  below'        ::      "[i,i,i,i,i,i,i,i,i]=>i"
  "inv"         ::      "i=>i"     ("(_~)" [70]  70)
  Fst,Snd       ::      "[i,i]=>i"
  below         ::      "[i,i]=>i" ("(_ ||/ _)" [67,66] 66 )
  beside        ::      "[i,i]=>i" ("(_ --/ _)" [67,66] 66 )

defs
  inv'_def   "inv'(A,B,R)== lwir(B,A);;[[Id(B),[[R,Id(A)]]]];;rwir(B,A)"
  inv_def    "R~         == inv'(dtyp(R),rtyp(R),R)"
  Fst_def    "Fst(A,R)   == [[R,Id(A)]]"
  Snd_def    "Snd(A,R)   == [[Id(A),R]]"
  below'_def "below'(A,B,C,De,E,F,G,R,S)  ==
                           reorg(A,E,F);;Snd(A,S);;(reorg(A,B,G)~);;
                           Fst(G,R) ;; reorg(C,De,G)"
  below_def  "R || S     == below'(ddtyp(R),rdtyp(R),drtyp(R),rrtyp(R),
                                   ddtyp(S),rdtyp(S),rrtyp(S),R,S)"
  beside_def "R -- S     == ((R)~ || (S)~)~"
end
```

Figure 12: Definition of simple combinators

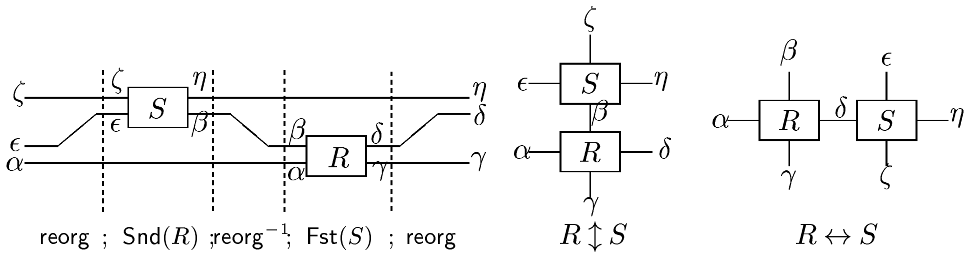

Figure 13: Graphical interpretation of below and beside

|| has the expected mathematical properties in terms of data values:

```
below_type  [|R :A*B<~>C*De;  S:E*F<~>B*G|]==> R || S:(A*E)*F<~>C*(De*G)
belowR      [|A:ChTy; B:ChTy; C:ChTy; De:ChTy; E:ChTy; F:ChTy; G:ChTy;
              R :A*B<R>C*De;  S:E*F<R>B*G|]==> R || S:(A*E)*F<R>C*(De*G)
belowI      [| a:sig(A); c:sig(C); d:sig(De); e:sig(E); f:sig(F);
               g:sig(G); <<a#b>,<c#d>>:R; <<e#f>,<b#g>>:S; b:sig(B)|]==>
                 < <<a#e>#f>,<c#<d#g>> > : R || S
belowE      [| < <<a#e>#f>,<c#<d#g>> > : R || S ;
               !!b.[| <<a#b>,<c#d>>:R; <<e#f>,<b#g>>:S; b:sig(B)|]==>P;
               R:A*B<~>C*De; S:E*F<~>B*H; a:sig(A'); c:sig(C');
               d:sig(De'); e:sig(E'); f:sig(F'); g:sig(G')|] ==> P
below_iff   [| a:sig(A');c:sig(C');d:sig(De');e:sig(E');f:sig(F');
               g:sig(G'); R:A*B<~>C*De;  S:E*F<~>B*G |] ==>
                 < <<a#e>#f>,<c#<d#g>> > : R || S  <->
                 (EX b:sig(B).<<a#b>,<c#d>> :R & <<e#f>,<b#g>> :S)
```

The combinator $\leftrightarrow$ (*beside*, which is written as `--` in Isabelle) is also depicted in Figure 13 and can be defined in terms of below and inverse. We prove that it has the expected mathematical properties:

```
beside_iff   [| a:sig(A'); b:sig(B'); c:sig(C'); e:sig(E'); f:sig(F');
                g:sig(G'); R:A*B<~>C*De; S:De*E<~>F*G  |] ==>
                  < <a#<b#e>>,<<c#f>#g> > : R -- S <->
                  (EX d:sig(De).<<a#b>,<c#d>> :R & <<d#e>,<f#g>> :S)
```

The Ruby relations 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. Most equality rules concerning simple combinators can be proved automatically by the tactic `ProveSimp`, which is similar to the one used to prove `assoc_comp` in Section 4.3.1:

```
fun ProveSimp prems =
     prove_equal prems 1
     THEN ALLGOALS (fast_tac_t RubyZF_cs prems);
```

where `RubyZF_cs` contains all introduction and elimination rules for the simple combinators and wiring relations and additionally all type rules have been added to `basictypeinfo`. Equations proved by above tactic are for example:

```
compinv     [| R:A<~>B; S:B<~>C |]==> (R ;; S)~ = S~;;R~
NNILinv     NNIL(A,B)~ = NNIL(B,A)
fstsndpar   [| R:A<~>B; S:C<~>De |]==> Fst(C,R);;Snd(B,S) = [[R,S]]
abides      [| R:A*B<~>C*De; S:De*E<~>F*G; T:H*I<~>B*J; U:J*K<~>E*L |]==>
                 (R -- S)||(T -- U)  = (R||T)--(S||U)
```

To demonstrate the use of the Pure Ruby induction theorem we sketch a proof of the so-called retiming property of Pure Ruby relations. The retiming property states that the absolute time can be changed without affecting the behaviour of the circuit. In Ruby this can be expressed by surrounding a relation with a delay and an inverse delay element:

```
- val prems = goal SimpComb.thy
    "!!R.R:A<R>B ==> R = D(A) ;; R ;; D(B)~";
Level 0
 1. !!R. R : A<R>B ==> R = D(A) ;; R ;; D(B)~
```

Applying the rule `rubyrel_induct` leads to the four subgoals below (one for each Pure Ruby element) which are easily proved. The actual proofs are not shown here:

```
- by (eresolve_tac [rubyrel_induct] 1);
Level 1
 1. !!R f A B. f : A~B ==> spread(f) = D(A) ;; spread(f) ;; D(B)~
 2. !!R A. D(A) = D(A) ;; D(A) ;; D(A)~
 3. !!R Ra S A B1 B2 C.
       [| Ra : A<R>B1; S : B2<R>C; Ra = D(A) ;; Ra ;; D(B1)~;
          S = D(B2) ;; S ;; D(C)~ |] ==>
       Ra ;; S = D(A) ;; (Ra ;; S) ;; D(C)~
 4. !!R Ra S A1 A2 B1 B2.
       [| Ra : A1<R>B1; S : A2<R>B2; Ra = D(A1) ;; Ra ;; D(B1)~;
          S = D(A2) ;; S ;; D(B2)~ |] ==>
       [[Ra,S]] = D(A1 * A2) ;; [[Ra,S]] ;; D(B1 * B2)~
```

5.3 Recursive Combinators

This section shows examples of recursively defined combinators, typically used to describe circuits with repetitive structures. Figure 14(a) illustrates how a combinator, mapf, which maps a relation to each element of a list, recursively can be defined in terms of Pure Ruby. The suffix f reflects that the argument relation of mapf is a function from its position in the structure to a signal relation. Figure 14(b) depicts the conventional graphical interpretation of mapf.

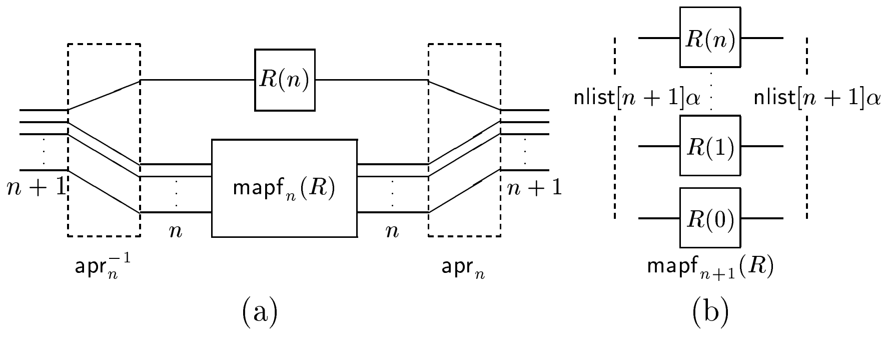

Figure 14: Graphical interpretation of mapf_{n+1}.

The type-parameterised version of map, `mapf'`, is defined in Isabelle using the primitive recursion operator over natural numbers, `rec`. It is shown in Figure 15 and follows the figure above. As for the simple combinators we define a version of map, `mapf`, where the type information is deduced from `R`. We prove that they describe the same relation:

```
mapf'_iff    [| n:nat; R:nat->A<~>B |] ==> mapf(n,R) = mapf'(A,B,n,R)
```

The conventional rules are proved for the mapf combinator. In particular `mapfR` shows that $\mathsf{mapf}_n(R)$ is a Pure Ruby relation if R is, and the two theorems `mapf_zero` and `mapf_succ`

```
RecComb = SimpComb +
consts
  mapf            ::  "[i,i]=>i"
  colf,rowf       ::  "[i,i,i]=>i"
  mapf'           ::  "[i,i,i,i]=>i"
  colf'           ::  "[i,i,i,i,i]=>i"

defs
  mapf'_def "mapf'(A,B,n,R) ==
                rec(n,NNIL(A,B),
                      %x y. (apr(A,x)~);; [[y,R`x]] ;; apr(B,x))"
  mapf_def  "mapf(n,R)== mapf'(dtyp(Union(range(R))),
                             rtyp(Union(range(R))),n,R)"
  colf'_def "colf'(A,B,C,n,R) ==
                rec(n,Fst(B,NNIL(A,C)) ;; cross(nlist[0]C,B),
                      %x y.Fst(B,apr(A,x)~) ;; (y || (R`x))  ;;
                           Snd(B,apr(C,x)))"
  colf_def  "colf(B,n,R) == colf'(ddtyp(Union(range(R))),B,
                                 rrtyp(Union(range(R))),n,R)"
  rowf_def  "rowf(B,n,R) == (colf(B,n,lam m:nat.((R`m)~)  ))~"
end
```

Figure 15: Definition of recursive combinators

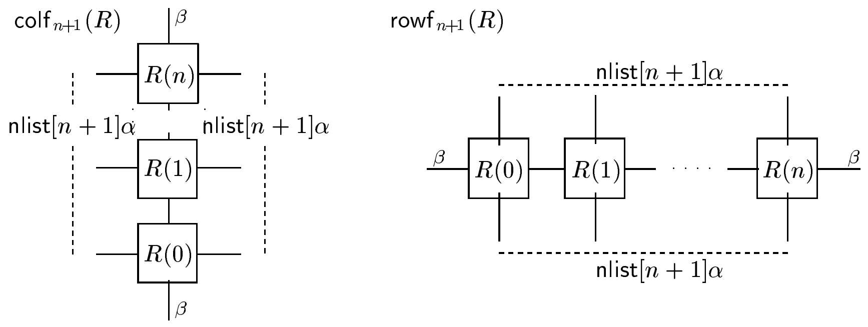

Figure 16: Graphical interpretation of colf$_{n+1}$ and rowf$_{n+1}$.

express that mapf enjoys the expected mathematical properties.

```
mapf_type  [| n:nat; R:nat -> A<~>B|]==> mapf(n,R): nlist[n]A<~>nlist[n]B
mapfR      [| n:nat; R:nat -> A<R>B|]==> mapf(n,R): nlist[n]A<R>nlist[n]B
mapf_zero  <snil,snil>: mapf(0,R)
mapf_succ  [| n:nat; R:nat->A<~>B; a:sig(A); b:sig(B);
              la:sig(nlist[n]A); lb:sig(nlist[n]B) |] ==>
                <[la<@a|n],[lb<@b|n]> : mapf(succ(n),R) <->
                  (<a,b>:R`n & <la,lb>: mapf(n,R))
```

Similarly the column and row structures depicted in Figure 16 can be defined in terms of Pure Ruby. Their definition in Isabelle is given in Figure 15 and the expected rules can be proved. For example the simplification rules for colf:

```
colf_zero  b:sig(B) ==> < <snil#b>,<b#snil> > :colf(B,0,R)
colf_succ  [|n:nat; R:nat->A*B<~>B*C; a:sig(A'); c:sig(C'); bs:sig(B);
             b0:sig(B); la:sig(nlist[n]A');lc:sig(nlist[n]C')|] ==>
           < <[la<@a|n]#bs>,<b0#[lc<@c|n]> >: colf(succ(n),R) <->
             (EX bn:sig(B).(<<a#bs>,<bn#c>>:R`n &
                           <<la#bn>,<b0#lc>>:colf(n,R)))
```

Note that in the case of $\mathsf{colf}_n(R)$ the type B cannot be deduced from the relation, since colf is not limited by the relation R in the zero case. Choosing the second part of the domain side of R would be too restrictive in the zero case. Thus we could not prove that the notational shorthand colf were equal to the full parameterised definition colf'.

6 Conclusion

ZF proved to be a well-suited basis for a Ruby formalisation. The Pure Ruby relations could easily be embedded, their types represented and the Pure Ruby set could be characterised by an inductive definition. The whole development is made as a conservative extension of ZF which means that it is both consistent and sound. The explicit type parameters, especially for wiring relations, makes the RubyZF notation a little clumsy, however, types can always easily be deduced from the context. Furthermore we demonstrated that for most combinators (e.g. relational inverse) a notational shorthand can be defined where the types can be deduced from the relational arguments. Due to the simplicity (purity) of ZF the system can serve as a reference implementation of Ruby, or in other words as a "lynchpin" for implementations of Ruby within other formalisms.

The development of specialised tactics in connection with type checking considerably increases the productivity and clarity in doing the proofs. The user is very seldom bothered with subgoals relating to type checking as this is all done behind the scenes. 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.

The parser and pretty printing mechanisms allowed us to obtain a syntax very close to the conventional Ruby syntax. However working in a huge theory like ZF it would be useful to be able to overwrite previous definitions for example such that parallel composition could be written with single square brackets.

The development of new theories is an interactive process where the definitions, even in underlying theories, are often changed. Our experience clearly shows that the use of general rule lists and automatic proof tactics means that most proofs can be left unchanged. So even if proofs are initially developed using low-level tactics it pays off to construct a high-level proof later.

7 Acknowledgements

Thanks to Robin Sharp for inspiring cooperation with the development of the T-Ruby design system, to Larry Paulson for being very helpful answering my Isabelle/ZF questions, and finally to Michael Mendler for showing interest in my project and for many valuable comments.

References

[1] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[2] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design.* Elsevier Publishers, 1990.

[3] 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.

[4] Lawrence C. Paulson. Set theory for verification: I. from foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993.

[5] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 148–161, Nancy, France, June 1994. Springer-Verlag LNAI 814.

[6] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.

[7] Lars Rossen. Proving (facts about) ruby. In G. Birtwistle, editor, *The IV Higher Order Workshop*, volume Workshops in Computing, pages Klüwer Academic Pub265–283. Springer Verlag Workshop in Computing, 1990.

[8] Lars Rossen. Ruby algebra. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits, Oxford 1990*, Workshops in Computing, pages 297–312. Springer-Verlag Workshop in Computing, 1991.

[9] Robin Sharp and Ole Rasmussen. Transformational rewriting with Ruby. In *CHDL '93*, pages 231–248. Elsevier Science Publishers (North-Holland), 1993.

[10] Robin Sharp and Ole Rasmussen. Using a language of functions and relations for VLSI specification. In *Functional programming and Computer Architecture, FPCA'95*, pages 45–54, June 1995.