Rewriting with Constraints in T-Ruby

Robin Sharp and Ole Rasmussen

Dept. of Computer Science, Technical University of Denmark

Abstract. This paper describes a tool for use in user-directed synthesis of circuits specified using the relational VLSI description language Ruby. The synthesis method is based on syntactic rewriting of Ruby terms, combined with the introduction of constraints into the specification. The rewriting process is described in a meta-language based on the use of tactics and tacticals, which makes it possible to develop complex specialised strategies for the refinement of specifications.

1 Introduction

The relational VLSI specification language Ruby [4] forms a good basis for a transformational style of VLSI synthesis, in which an abstract specification is systematically modified into a description of an implementable circuit by "calculation" [5, 10].

This paper describes some of the principles behind a tool, known as T-Ruby [12], for handling this style of synthesis. In T-Ruby, the transformation process relies on two elements: *Syntactic rewriting*, based on (possibly conditional) equivalences between terms of the Ruby language [8], and the introduction of *constraints* related to data abstractions or specialisations. These constraints enable us to produce circuits which are no longer simply equivalent to the original specification, by reducing the generality of the circuit whose description is being developed. The transformation process takes place in a Standard ML environment in an interactive manner directed by the user. The availability within T-Ruby of a meta-language with tactics and tacticals makes it possible to compose complex rewriting strategies to make the transformation process more automatic.

The paper is not a complete description of T-Ruby. Many features are not covered fully, and some are not described at all. For further details, see [12].

2 The T-Ruby Language

Ruby is a language intended for specifying VLSI circuits in terms of relational abstractions of their behaviour. A circuit is described by a binary relation, and the language permits simple relations to be composed into more complex ones by the use of a variety of combining forms which are higher-order functions.

T-Ruby is based on a formalisation of Ruby as a language of functions and relations, with a conventional set of elementary types (integers, Booleans, bits, characters,...), and a set of type constructors for constructing constructed types,

such as the types of pairs, lists and binary relations. Superficially, it resembles a traditional strongly typed functional programming language with parametric polymorphism, offering facilities for defining named objects of these types and functions for manipulating them. However, it contains two slightly unusual features which reflect its true nature. Firstly, the language has built-in constants for certain basic relations and operators which are needed for VLSI design. And secondly, the language extends the normal function space for functions to include dependent product types [6, 15].

The built-in relational constants are those of the so-called Pure Ruby subset of Ruby, as introduced by Rossen [11]. 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 combinators. These are usually defined in terms of synchronous streams of data by the following four axioms:

$$a \ \mathsf{spread}\, \mathbf{r} \ b \triangleq \forall t \cdot a(t) \ \mathbf{r} \ b(t) \tag{1}$$

$$a \ \mathcal{D} \ b \triangleq \forall t \cdot a(t) = b(t+1) \tag{2}$$

$$a \ F \,;\, G \ b \triangleq \exists c \cdot (a\, F\, c \ \wedge \ c\, G\, b) \tag{3}$$

$$<a_1, a_2> \ [F, G] \ <b_1, b_2> \ \triangleq a_1\, F\, b_1 \ \wedge \ a_2\, G\, b_2 \tag{4}$$

The classical view of these is that the relation $\mathsf{spread}\, r$ describes (any) combinational circuit, and the relation $\mathcal{D}$ describes the basic sequential circuit—the so-called *delay* element. $F; G$ (the backward relational composition of F with G) describes the serial composition of the circuit described by F with that described by G, and $[F, G]$ (the relational product of F and G) correspondingly describes the parallel composition of F and G.

2.1 Graphical Interpretations

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. Since this can be a great help to the intuition, we remind the reader that 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. Boxes can be drawn in a "2-sided" or "4-sided" manner. A 2-sided box is drawn with the components of the domain up the left hand side and the components of the range up the right. On a 4-sided box, the components of the domain must always be a pair, whose first element is drawn up the left hand side and second element along the top from left to right, while the range is drawn similarly along the bottom from left to right, and up the right hand side.

The conventional graphical interpretation for $\mathcal{D}$ is as a D-shaped figure, with the domain up the flat side and the range up the rounded side, while F ; G is drawn with the range of F "plugged into" the domain of G, and $[F, G]$ with the

two circuits F and G in parallel (unconnected). These conventions are illustrated in Figure 1. For further details, see [4].

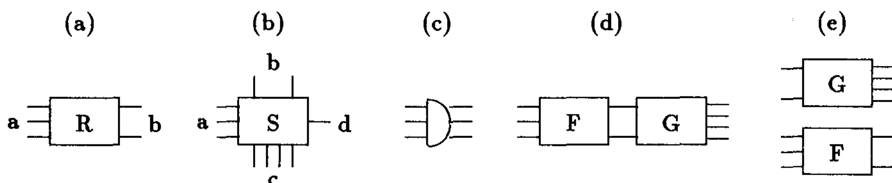

Fig. 1. Graphical interpretations of:

(a) $R =$ spread r_1 in 2-sided version, with domain a and range b,
(b) $S =$ spread r_2 in 4-sided version, with domain $<a, b>$ and range $<c, d>$,
(c) $\mathcal{D}$,
(d) $F\,;G$,
(e) $[F, G]$

2.2 Circuit and Combinator Definitions

In T-Ruby, all other Ruby circuits and combinators are defined in terms of the four elements of Pure Ruby, using a simple definition syntax in the style of the typed λ-calculus [1]. Commonly used simple definitions which we shall make use of in the examples of this paper are shown in Figure 2. Note that in T-Ruby the combinational relation r which is used as the argument to spread is represented by r's characteristic function. In the concrete syntax used in the figure, notations of the form `\x:t.b` stand for λ-abstractions with bound variable x of type t, `'A, 'B,...` stand for type variables, and notations of the form `ty1*ty2` and `ty1~ty2` stand respectively for pair types and relation types with component types `ty1` and `ty2`.

Circuits are by definition non-parameterised relations. Thus `id` describes the identity relation, often denoted ι, Dub relates a value to a pair of copies of this value, Cross relates a pair of values to the swapped pair, and reorg rearranges the way in which three values are grouped into pairs. Combinators have one or more parameters, corresponding for example to the circuits to be combined; applying a combinator to suitable arguments gives a new circuit. Thus (Fst R) is the circuit described by $[R, \iota]$, and so on. The graphical interpretations of these relations are shown in Figure 3.

2.3 Size Parameterisation

The dialect of Ruby used by the system is essentially that given by Rossen in [9]. This differs from the standard description of Ruby given by Jones and Sheeran

```
      circuit(2) id    = spread(\a:'A, b:'A.(a = b));;
      circuit(2) Dub   = spread(\a:'A, b:('A*'A).
                               (let (b1,b2) = b in
                               ((a=b1) & (a=b2)) ));;
      circuit(4) Cross = spread(\a:('A*'B), b:('B*'A).
                               (let (a1,a2) = a,
                                    (b1,b2) = b in
                               ((a1=b2) & (a2=b1)) ));;
      circuit(2) reorg = spread(\a:(('A*'B)*'C), b:('A*('B*'C)).
                               (let ((a1,a2),a3) = a,
                                    (b1,(b2,b3)) = b in
                               ((a1=b1) & (a2=b2) & (a3=b3)) ));;
      combinator Fst   = \H:('A~'B).([H,id]);;
      combinator Snd   = \H:('A~'B).([id,H]);;
      combinator loop4 = \H:(('A*'B)*'C)~('D*('E*'B)).
                               (Cross; loop2(reorg;Cross;H;reorg~));;
  rec combinator mapn  = \n:int, F:'A~'B.
                               (if n=0 then NNIL
                                else (apln (n-1))~;[F,(mapn (n-1) F)];
                                     (apln (n-1)) );;
```

Fig. 2. Simple circuit and combinator definitions in T-Ruby.

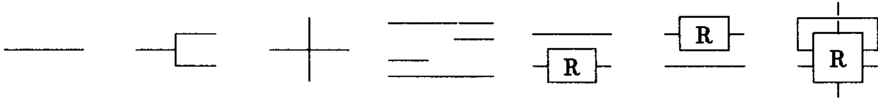

Fig. 3. Graphical interpretations of ι, Dub, Cross, reorg, (Fst R), (Snd R) and (loop4 R).

in [4] in that "repetitive" combinators and wiring relations are explicitly parameterised in the number of repetitions. Thus, for example, the wiring relation apl, which relates a list of signals, t, and a single signal, s, to the list formed by appending s "on the left" of t, becomes a combinator parameterised in the number of signals in t. To stress this parameterisation, the names of such combinators are changed by adding the letter n; thus apl becomes apln. Similarly, map (which applies the same relation to all elements of a list) becomes mapn; applied to an integer argument m and a relation R of type $(ty1 \sim ty2)$, this gives an object of type:

$$(\mathsf{nlist}[m]ty1 \sim \mathsf{nlist}[m]ty2)$$

i.e. a relation between a list of exactly m elements of type $ty1$ and a list of m elements of type $ty2$.

Parameterisation of combinators and wiring relations in the size of the cir-

cuits involved has two beneficial effects. Firstly, it means that instantiation of circuits of particular sizes is made explicit in T-Ruby expressions, and secondly, it eases the task of proving facts about circuits by using a theorem prover [11]. However, it also has the consequence that the result type of functions will often depend on the value of the argument supplied to the function, so the ordinary function spaces must be generalised to the *dependent product* types introduced by Martin-Löf [6, 15]. A comparatively simple example is the case of the combinator mapn above. This has the polymorphic dependent product type:

$$n : \mathsf{int} \rightarrow ((\alpha \sim \beta) \rightarrow (\mathsf{nlist}[n]\alpha \sim \mathsf{nlist}[n]\beta))$$

where the result type depends on the value of the first argument, here denoted n, and α and β are type variables.

In general, dependent product types make type inference undecidable and introduce a number of complications into the type unification used for type checking. These are described in more detail in [13, 12].

3 Rewriting Ruby Terms

The T-Ruby system allows the user to rewrite Ruby terms according to predefined rewrite rules. Rewriting takes place in an interactive manner directed by the user. This style of system is often called a *transformation system* to distinguish it from a conventional rewrite system. The system uses a meta-language to describe the desired transformations in terms of a series of transformation commands. This allows the user to design his or her own transformations and to remain in full control of the process. At the same time, the meta-program serves as documentation for the transformation performed.

This approach is inspired by work of Milner, Paulson and others on LCF [3, 7]. There are a number of basic rewrite functions, known as *tactics*, which can be combined by the use of higher order functions, known as *tacticals*. In the T-Ruby system, these functions are Standard ML functions, and are applied in an SML environment. The basic functions are quite simple, which means that the system itself cannot perform any sophisticated rewrites. However, as the user becomes familiar with the domain on which the system is used, more complex and automated strategies can be designed via the use of tacticals.

3.1 The Transformation Process

The basic idea of the rewrite system is to rewrite Terms (or subterms within Terms), using a sequence of directed rewrite rules. In what follows, we shall refer to the Term which is currently to be rewritten as the *target expression*.

We can write a directed rule in the form:

$$l \;\rightarrow\; r$$

where l and r are in general Terms, possibly with free variables. Such a directed rule indicates that any (sub-)Term, say t, which *matches* l can be replaced by

r. Matching will in general involve *instantiation* of type variables or free term variables within l, and in such a case the same instantiations must be applied to r before it replaces l. Formally speaking, the matching process involves finding a *substitution*, σ, which can be applied to the (type and term) variables of l to make it the same as t: i.e. such that $\sigma l = t$.

When manipulating Ruby expressions with a view to producing a design, we essentially start from the trivial equivalence:

$$spec = spec$$

where *spec* is the initial specification of a circuit at some suitably high level of abstraction, in T-Ruby known as the *start expression*. We then attempt to rewrite the right-hand side by using proved rewrite rules, so that we arrive at a description which is closer to what we can implement. However, if we restrict our attention to rewriting with simple equivalences the result will be strictly equivalent to the original abstract description, and will therefore often be impossible to realise. For example, it might be expressed in terms of integers; to realise the circuit, we have to restrict ourselves to integers representable by n bits, and so on.

Instead of just rewriting the right-hand side of the trivial equivalence, we are therefore interested in performing a calculation where the specification can also be manipulated. As pointed out in [14], we can illustrate this process by:

$$\begin{array}{rcl} spec & = & spec \\ & & \downarrow \mathcal{R}_1 \\ spec & = & step_1 \\ \mathcal{C}_1 \downarrow & & \downarrow \mathcal{C}_1 \\ spec' & = & step_1' \\ & & \downarrow \mathcal{R}_2 \\ spec' & = & step_2 \\ \vdots & & \vdots \\ spec''^{\cdots\prime} & = & impl \end{array}$$

where $\mathcal{R}_i$ represent sequences of rewrites, which are only applied to the right-hand side of the equivalence, and $\mathcal{C}_i$ represent *constraints*, which are applied to both sides of the equivalence. This style of design has been described by Rossen in [10]. What we see as users of the rewrite system is, of course, just the right-hand side of this process:

$$spec \rightarrow step_1 \rightarrow step_1' \rightarrow step_2 \rightarrow \ldots \rightarrow impl$$

but behind the scenes the original specification is changed from *spec* to $spec''^{\cdots\prime}$, reflecting the addition of the constraints. From a logical point of view, it is equivalent to demonstrating that the implementation and specification are equivalent under the constraints [16], i.e. that:

$$constraints \vdash (impl \Leftrightarrow spec)$$

The T-Ruby system permits the user to inspect the *start expression*, together with the currently added constraints, at any stage of the rewriting process.

3.2 Rewrite Rules

In the T-Ruby system, all circuits, combinators and rules are represented as Terms composed from free or bound variables, function applications, quantifications, conditionals and functional abstractions. A rule must be a Term representing an equality or an implication between two equalities, with universal quantification over variables. Variables which appear on the left-hand side of the rule must also appear on the right, in order to satisfy the stability criterion for rewriting [2], and this implies that rewriting alone cannot introduce new variables into a Term. Apart from this there are no restrictions on the forms of the rules which may be used. In practice, however, most of the commonly used rules are equalities between relations, corresponding to equivalences between circuits, which can be used to manipulate a circuit description in Ruby to another, equivalent form.

Some simple examples which are used later in the paper can be seen in Figure 4. These rules express facts about the combinators, such as the associativity of serial composition (assoccomp), the commutativity of Fst and Snd (fstsndcomm), and the distributivity of Fst over serial composition (fstcompdist). More complex rules are used in Ruby synthesis to express such things as the input-output equivalence of a circuit and a systolic version of the same circuit, or to express ways to replace complex wiring with simpler equivalents.

$$\begin{array}{ll}
\mathsf{assoccomp} & \triangleq \forall R:\alpha\sim\beta,\ S:\beta\sim\gamma,\ T:\gamma\sim\delta\cdot(\ (R\,;S)\,;T\ =\ R\,;(S\,;T)\ )\\
\mathsf{fstsndcomm} & \triangleq \forall R:\alpha\sim\beta,\ S:\gamma\sim\delta\cdot(\ (\mathsf{Fst}\ R)\,;(\mathsf{Snd}\ S)\ =\ (\mathsf{Snd}\ S)\,;(\mathsf{Fst}\ R)\ )\\
\mathsf{fstcompdist} & \triangleq \forall R:\alpha\sim\beta,\ S:\beta\sim\gamma\cdot(\ \mathsf{Fst}\,(R\,;S)\ =\ (\mathsf{Fst}\ R)\,;(\mathsf{Fst}\ S)\ )\\
\mathsf{parcompdist} & \triangleq \forall R:\alpha\sim\beta,\ S:\gamma\sim\delta,\ T:\beta\sim\epsilon,\ U:\delta\sim\zeta\cdot\\
 & \qquad (\ [R,S]\,;[T,U]\ =\ [R\,;T,\ S\,;U]\ )\\
\mathsf{idcompl} & \triangleq \forall R:\alpha\sim\beta\cdot(\ R\ =\ \iota\,;R\ )\\
\mathsf{loop4comm} & \triangleq \forall R:\zeta\sim\beta,\ S:(\alpha\times\beta)\times\gamma\sim\delta\times(\epsilon\times\zeta)\cdot\\
 & \qquad (\ \mathsf{loop4}\ (S\,;\mathsf{Snd}\ (\mathsf{Snd}\ R))\ =\ \mathsf{loop4}\ (\mathsf{Fst}\ (\mathsf{Snd}\ R)\,;S)\ )
\end{array}$$

Fig. 4. Simple rewrite rules in T-Ruby.

In the T-Ruby system, the directed rules used for rewriting come from four sources:

1. They may be defined explicitly via *rewrite rule* definitions. Such rewrite rules have the form of equations, possibly with preconditions, and therefore define two directed rules, one from left to right, and one from right to left

in the equation. For example the rewrite rule assoccomp in Figure 4, which expresses the associativity of serial composition, gives rise to the two directed rules:

$$\forall R:\alpha \sim \beta,\ S:\beta \sim \gamma,\ T:\gamma \sim \delta \cdot ((R\,;S)\,;T \ \rightarrow\ R\,;(S\,;T))$$
$$\forall R:\alpha \sim \beta,\ S:\beta \sim \gamma,\ T:\gamma \sim \delta \cdot (R\,;(S\,;T) \ \rightarrow\ (R\,;S)\,;T)$$

We will refer to these as the *introduction* and *elimination* rules respectively.

2. They may be defined implicitly via *circuit* definitions. For example, the definition of the circuit Dub shown in Figure 2 above gives rise to the introduction rule:

$$\mathsf{Dub} \ \rightarrow\ \mathsf{spread}(\lambda\, a:\alpha, b:(\alpha \times \alpha) \cdot (\mathsf{let}\ (b1, b2) = b\ \mathsf{in}\ ((a = b1) \wedge (a = b2))))$$

which permits the named circuit to be expanded with its definition, and an elimination rule which goes in the reverse direction.

3. They may be defined implicitly via *combinator* definitions. For example, the definition of the combinator Fst gives rise to the introduction rule:

$$\forall H:\alpha \sim \beta \cdot (\mathsf{Fst}\ H \ \rightarrow\ [H, \iota])$$

which permits the named combinator to be expanded with its definition (with universal quantification over parameters of the combinator), and an elimination rule which goes in the reverse direction.

4. They may be defined as the results of previous rewrite processes. If a sequence of rewrites has enabled us to rewrite a Term t to another Term t', then the rule $t = t'$, universally quantified over all free variables in t, can be stored in the system. It may be helpful to think of this as a *rewrite lemma* which can be stored for later use.

For a rewrite rule, lemma, circuit or combinator definition with identifier nam, the introduction and elimination rules will have the identifiers nam-i, and nam-e respectively.

3.3 Proof Obligations

Explicitly defined rewrite rules are normally proved from the axioms of Pure Ruby before being entered into the system. However, the existence of a proof is not checked by the system and therefore unproved rewrite rules can be introduced at any time. This is useful if we are convinced of the existence of a useful rule which is not yet in the system, but it offers the potential danger that the rewrite process will not be sound. To enable the user to ensure the correctness of the rewrite process, all the unproved rewrite rules used are printed out in instantiated form at the end of the process, together with the names of the original rules from which they have been instantiated. These rules form a proof obligation for the rewrite process.

A particular problem of correctness occurs when dealing with conditional rewriting. Ideally, each time a match was found with the left hand side of a conditional rewrite rule, the instantiated precondition would automatically be proved by a theorem prover. This is not at present possible, as proving tools are not fully automatic and are far too slow for use as co-routines for another process. The main goal of the rewrite process would be lost in the effort of performing the proof. Again, the solution in the T-Ruby system is to print out the conditional rules in instantiated form and say that they are part of the proof obligation for the rewrite process.

4 The Rewrite Process

The rewrite process falls into three phases: The initial phase, the rewrite phase and the final phase. In the initial phase the start expression is entered, possibly together with some specialised rules for use in the transformation. In the rewrite phase, actual rewriting is performed by applying suitable tactics to the target expression, and constraints can be added to the target expression. The rewrite session ends in the final phase where the system prints out three things: The start expression with added constraints, the final target expression, i.e. the result of the rewrite process, and finally the list of all the unproved rewrite rules used in the rewrite process. Some complete examples can be seen in [14, 12].

4.1 Search Strategies and Selection

Four strategies are available for performing the basic rewrite steps, offering different ways to search the term for a match: Two use bottom-up search in the tree which represents the Term (*left-depth, right-depth*), and two use top-down search (*left-top, right-top*). This does not extend the functionality of the system, since the rewrite system contains a *backtracking* mechanism which allows the user to go from one successful match for a given rule to the next one, in the order defined by the strategy in use. Thus the system will eventually find all possible matches, regardless of the strategy chosen, but of course the strategy in use determines how many attempts are required to find a particular match.

Another way to "navigate" within a term to be rewritten is to use *selection*. This enables the user to choose a subterm from the target expression and continue to rewrite on that subterm. It changes the state of the rewrite system so that the subterm is made the new target expression. When rewriting on the subterm has finished, the state can be changed back to consider the full term, where the rewritten subterm is inserted in the place from which it was removed.

In the T-Ruby system, selection is performed by pattern matching. The user provides a term with free variables, which is to act as a pattern. The system will try to find a subterm which matches this pattern (via instantiation of some or all of the free variables). If this subterm is not the desired one, it is possible to backtrack and find the next match for the same pattern. To allow for experiments it is also possible interactively to *undo* previous rewrite steps and constraints completely.

4.2 Rewrite Tactics and Tacticals

Rewrite tactics are functions which can be applied to Terms, and which attempt to perform a rewrite according to specific rules for matching sub-terms. When applied to a Term, the tactic may *succeed*, in which case the target expression is rewritten in the way prescribed by the tactic, or it may *fail*, in which case the target expression is left unchanged.

The T-Ruby system offers a set of four basic functions for constructing basic tactics which apply named rewrite rules using a particular search strategy in the target expression. For example the function `Rd`, applied as:

```
Rd ["assoccomp-i","fstcompdist-i","parcompdist-i"]
```

gives a rewrite tactic which will rewrite the target expression once, using a rightmost depth-first search strategy, and attempting to apply rules `assoccomp-i`, `fstcompdist-i` and `parcompdist-i` until one of them succeeds. Similarly, `Rt`, `Ld` and `Lt` give a tactic which uses *right-top*, *left-depth* and *left-top* search respectively.

Rewrite tactics can be combined in a conjunctive, disjunctive or repetitive manner by the use of higher-order functions, known as *tacticals*, to form new and more complex rewrite tactics. The two basic tacticals are the infix operators `THEN` and `ORELSE`. They combine tactics sequentially and alternatively and make use of an automatic backtracking mechanism to find possible solutions. Automatic backtracking enables us to write more general complex rewrite tactics, thus building up a library of tactics for different tasks.

The semantics of the conjunctive tactical `THEN`, applied as `tac1 THEN tac2`, can be described in a simple way as follows: It first attempts to apply `tac1` and then, if this succeeds, to apply `tac2`. More exactly, the system automatically applies backtracking on the list of solutions produced by `tac1` until a solution for which `tac2` succeeds is found. This solution is presented to the user, who can then use explicit backtracking to find further solutions, if any. If there are no solutions where `tac2` succeeds in the list produced by `tac1`, then the combined tactic fails.

The disjunctive tactical `ORELSE`, applied as `tac1 ORELSE tac2`, produces a list of all solutions from `tac1` followed by all solutions from `tac2`. The combined tactic fails if both `tac1` and `tac2` fail.

The tactical `REPEAT tac1` repeatedly applies `tac1` until it fails. An extended version of `REPEAT` is the tactical `REPEAT_FIRST tac1 tac2` which first applies `tac1`, and then, if that fails, applies `tac2`. This procedure is repeated until `tac1` succeeds or `tac2` fails. Both tacticals succeed if and only if `tac1` succeeds at least once. They are both based on the tactical `THEN` and therefore use the same kind of automatic backtracking to find possible solutions. This makes `REPEAT_FIRST` extremely powerful for automatically searching through all possible solutions to make a certain rule match. In practical examples it has been particularly useful when we wish to exploit the associativity of operators such as serial composition until some other rule matches (see example below).

4.3 Composed Tactics

The use of tacticals enables us to write specialised composed tactics for specific rewrite tasks. Some of these are of a "technical" nature, reflecting properties of the Ruby language, while others represent more or less complex structural manipulations of the circuit, and thus correspond to more traditional design steps. We have only space for a very few, simple examples.

A "technical" tactic which lets us use a rewrite rule (plusid) without bothering about the parenthesis structure for the serial composition operator is shown below. It first moves all the parentheses as far as possible to the right and then moves them left until the rule matches. Typically we would write an SML function with the rule name (here plusid) as a parameter, thus having a specialised tactic for easy rewriting of expressions with complex serial compositions.

```
(* Given the target expression:  *)
  (([x2,x2];(plus;(plus)~));(plus;x2))

(* and the rule for 'plus' that: *)
  rule plusid = ((plus~;plus) = (idZ));;

(* Rewrite by the tactic:        *)
- by ( (REPEAT (Ld ["assoccomp-i"]) )
=     THEN
=       (REPEAT_FIRST (Ld ["plusid-i"]) (Ld ["assoccomp-e"])) );

(* This produces the result:     *)
  ([x2,x2];(plus;(idZ;x2)))
```

Other (more complex) tactics in a similar style can be constructed for replacing, say, interleavings of wires with less complex wiring structures, and performing other design steps of this nature.

In T-Ruby, tactics are also available for reduction of arithmetic and Boolean Terms, so that the user does not need to apply individual rules for this purpose. For example, the tactic Evaluate evaluates all evaluable integer and Boolean sub-terms in the current target expression, where a sub-term is considered evaluable if it contains no uninstantiated variables, and no sub-terms of types other than integer or Boolean, while Full_evaluate also performs ordinary β-reduction for function applications of any type.

Evaluation makes it, for example, possible to rewrite an expression into Pure Ruby by expanding all the combinators and circuits by their definitions. After each expansion, all if- and λ-expressions are reduced with Full_evaluate to ensure that no infinite expansions arise. For example, a tactic for expanding mapn into Pure Ruby is:

```
- by (REPEAT (      (Ld ["mapn-i","apln-i","NNIL-i"])
=               THEN  Full_evaluate ) );
```

From the target expression (`mapn 2 plus`), this produces the Pure Ruby Term:

```
(((spread(\a:((int*int)*nlist[1](int*int)).(\b:nlist[1+1](int*int).
(b=(ncons 1 fst a snd a)))))~;[plus,
(((spread(\a:((int*int)*nlist[0](int*int)).(\b:nlist[0+1](int*int).
(b=(ncons 0 fst a snd a)))))~;[plus,
spread(\a:nlist[0]'A.(\b:nlist[0]'B.(a=nnil & b=nnil)))]);
spread(\a:(int*nlist[0]int).(\b:nlist[0+1]int.
(b=(ncons 0 fst a snd a)))))]);
spread(\a:(int*nlist[1]int).(\b:nlist[1+1]int.
(b=(ncons 1 fst a snd a)))))
```

5 Constraints

In T-Ruby, a constraint can have one of two forms: it can either be a relational constraint or an instantiation, both of which restrict the specification to a more specific implementation.

5.1 Relational Constraints

A *relational constraint* is technically just an extra sub-term added to both sides of an equivalence. For example, given an equivalence between Ruby Terms t_1 and t_2:

$$t_1 = t_2$$

then the following is also an equivalence:

$$comb(con, t_1) = comb(con, t_2)$$

where *con* is the sub-term giving the constraint, and *comb* is an appropriate combinator for combining two sub-terms. Note that *con* can contain new free variables, and that this is the only way to introduce new parameters into a circuit description.

In many practical examples, *comb* is the combinator for serial composition, and the constraint is a representation relation or other type-constraining relation. Constraints can be introduced on the domain or range side of an expression representing a Ruby relation, so from an equivalence between two relations expressed by $S = I$, we can for example construct the valid equivalences:

$$\mathcal{C}_d \,;\, S = \mathcal{C}_d \,;\, I$$
$$S \,;\, \mathcal{C}_r = I \,;\, \mathcal{C}_r$$

where $\mathcal{C}_d$ and $\mathcal{C}_r$ are suitable sub-terms giving the constraints.

To illustrate this, let us consider a situation in the development of a simplified calculator, as described in some detail in [14]. The initial specification of this circuit is developed in terms of integers. Essentially, the calculator is to offer two functions: add, where the current integer input is to be added to the accumulated sum (unless overflow occurs), and pass, where the integer is to be used to initialise the sum. The initial part of the rewrite process proceeds as follows:

```
- new_rw "SUMspec";
SUMspec

- by (Ld ["SUMspec-i"]);
(loop4 (((Fst (Snd D));ALUa);(Snd Dub)))

- by (      (Ld ["lemmal-i"])
=      THEN (Ld ["ALUk-i"  ]));
(loop4 (((Fst (Snd D));((((Snd Decode);(Fst ((Fst Dub);reorg)));
          ((Cross|ADDa)--Mux));(Fst p1)));(Snd Dub)))
```

We start the rewrite process with new_rw "SUMspec", so that the term SUMspec becomes the initial target expression. SUMspec is then expanded with its definition by a (left-depth) rewrite. This gives the circuit shown in Figure 5(a). The abstract ALU sub-circuit ALUa is then replaced by an equivalent, more concrete sub-circuit ALUk, by use of lemmal, which merely states the (provable) fact that ALUa = ALUk. Finally, ALUk is expanded with its definition, to allow us to make rewrites on its subcomponents.

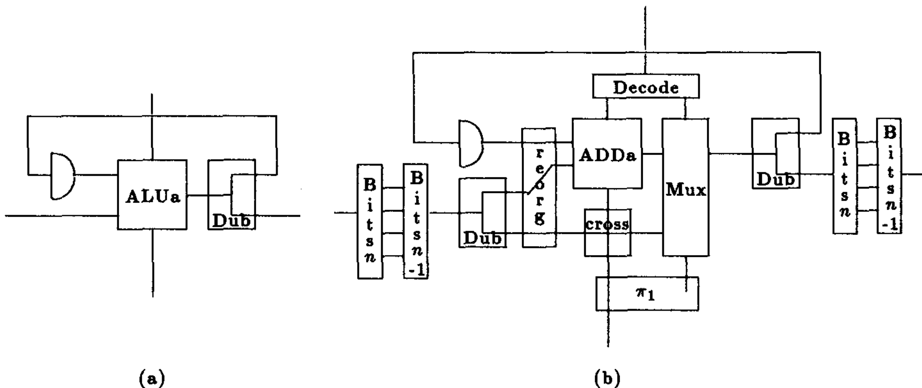

Fig. 5. Simple calculator.

(a) Original specification using SUMspec.
(b) After expansion of ALUk and addition of constraints.

Then, to restrict the interface of the calculator from being arbitrary integers to only being integers in an n-bit representation, we add suitable constraints to the domain (by applying the function dcon) and range (with rcon). A suitable constraint here is the identity relation on n-bit integers, denoted (idn n). Such an identity relation is constructed as the composition of the corresponding representation relation with its inverse [5]. The required representation relation

relates an integer which can be represented by n bits to the n bit values, and is here denoted Bitsn n. After introduction of the constraints and expansion of idn with its definition, we obtain the circuit shown in Figure 5(b).

```
- dcon "Fst (idn %n:int);";
The new expression is:
((Fst (idn %n:int));
 (loop4 (((Fst (Snd D));((((Snd Decode);(Fst ((Fst Dub);reorg)));
          ((Cross|ADDa)--Mux));(Fst p1)));(Snd Dub))))

- rcon ";Snd(idn %n:int) ";
The new expression is:
(((Fst (idn %n:int));
 (loop4 (((Fst (Snd D));((((Snd Decode);(Fst ((Fst Dub);reorg)));
          ((Cross|ADDa)--Mux));(Fst p1)));(Snd Dub))));
  (Snd (idn %n:int)))

- by (REPEAT (Ld ["idn-i"]));
(((Fst ((Bitsn %n:int);((Bitsn %n:int))~));
 (loop4 (((Fst (Snd D));((((Snd Decode);(Fst ((Fst Dub);reorg)));
          ((Cross|ADDa)--Mux));(Fst p1)));(Snd Dub))));
  (Snd ((Bitsn %n:int);((Bitsn %n:int))~)))
```

Note that adding these constraints introduces a free variable %n into the description, representing the number of bits in the integer representation.

5.2 Instantiation

The second form of constraint in T-Ruby is *instantiation* (or *specialisation*) of term or type variables on both sides of an equivalence, so that they take on particular values or types. Technically, instantiation in T-Ruby is the application of a pair of substitutions, $(\sigma_{term}, \sigma_{type})$, where σ_{term} is a mapping from free term variables or universally quantified variables to terms, and σ_{type} is a mapping from type variables (which are always free) to types.

For an instantiation to be valid when applied to a Term s, each substitution, say of variable x by Term t, must obey certain obvious rules:

1. The types of x and t must be unifiable.
2. The instantiated variable, x, must occur as a free or universally quantified variable in s, but *not* in t.
3. If t contains a variable y ($\neq x$), then y must occur in s. The type of y within t will then be unified with its type within s. This means, amongst other things, that a term substitution may implicitly give rise to a type substitution.

When the instantiation forms part of a rewriting process, these rules must of course be obeyed on both sides of the current equivalence, $spec^i = step_i$. As

in the case of relational constraints, inspection of the start expression will show the currently applied instantiations.

For example, the "n-bit" description of the calculator given above could be instantiated to give a description of a calculator which uses a 16-bit representation of integers by the following commands, which also demonstrates the result of the instantiation on the start expression (with its relational constraints).

```
- instantiate [("16","n")] [ ];
The term is instantiated to:
(((Fst ((Bitsn 16);((Bitsn 16))~));(loop4 (((Fst (Snd D));
      ((((Snd Decode);(Fst ((Fst Dub);reorg)));((Cross|ADDa)--Mux));
       (Fst p1)));(Snd Dub))));(Snd ((Bitsn 16);((Bitsn 16))~)))

- print_startexp();
(((Fst (idn 16));SUMspec);(Snd (idn 16)))
```

6 Conclusion

The style of transformational synthesis illustrated in this paper is useful for a large proportion of the typical areas of application of Ruby. The availability of the T-Ruby tool has made this style of design more secure by placing the transformations on a formal basis. The transformation process can be continued down to any desired level of detail, leading ultimately to a circuit description at the level of standard cells in a cell library.

T-Ruby offers three features which make it easy to perform powerful transformations. Firstly, many of the equivalences of Ruby are extremely powerful, in the sense that they express radical rearrangements of the circuit. Secondly, long sequences of rewrites can be stored as new rewrite rules, which can then be applied by using a simple tactic. And thirdly, it is possible to compose complex tactics by using tacticals. We are gradually building up a library of such tactics for general use.

Currently, there is no direct connection from the system to a theorem prover for handling the proof obligations which arise during the design process. Nor is there a direct link to a drafting system which can automatically draw the graphical interpretations of the circuit as the design progresses. Work is in progress to add these components to the system.

7 Acknowledgements

The work described in this paper has been partially supported by the Danish Technical Research Council as part of the *RapID* project on formal methods in computer science.

The authors would like to thank Lars Rossen for many interesting discussions about constructing tools for Ruby.

References

1. A. Church. *The Calculi of Lambda-conversion.* Princeton University Press, Princeton, New Jersey, 1941.
2. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Science, Volume B: Formal Models and Semantics,* chapter 6, pages 243–320. Elsevier Science Publishers B.V., 1990.
3. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation,* volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.
4. G. Jones and M. Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design,* pages 13–70. Elsevier Science Publishers B.V., 1990.
5. G. Jones and M. Sheeran. Relations and refinement in circuit design. In C. C. Morgan and J. C. P. Woodcock, editors, *Proceedings of the 3rd. BCS FACS Workshop on Refinement,* Workshops in Computing, pages 133–152, London, January 1991. BCS, Springer-Verlag.
6. P. Martin-Löf. Constructive mathematics and computer programming. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages,* pages 167–184. Prentice-Hall, London, 1985. Also published in *Proc. 6th. International Congress for Logic, Methodology and Philosophy of Science,* 153–175 (North-Holland, 1982).
7. L. C. Paulson. *Logic and Computation,* volume 2 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1987.
8. Ole Rasmussen. A Ruby rewrite system. Master's thesis, Dept. of Computer Science, Technical University of Denmark, February 1992.
9. L. Rossen. Formal Ruby. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design,* pages 179–190. Elsevier Science Publishers B.V., 1990.
10. L. Rossen. Proving (facts about) Ruby. In G. Birtwhistle, editor, *IV Higher Order Workshop, Banff,* Workshops in Computing, pages 265–283. Springer-Verlag, 1990.
11. L. Rossen. Ruby algebra. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits, Oxford 1990,* Workshops in Computing, pages 297–312. Springer-Verlag, 1990.
12. R. Sharp. T-Ruby: A tool for handling Ruby expressions. Technical Report ID–TR: 1992-112, Dept. of Computer Science, Technical University of Denmark, September 1992.
13. R. Sharp. The Ruby framework. Technical Report ID–TR: 1993-xx, Dept. of Computer Science, Technical University of Denmark, 1993. To appear.
14. R. Sharp and O. Rasmussen. Transformational rewriting with Ruby. In L. Claesen, editor, *CHDL'93.* IFIP WG10.2, Elsevier Science Publishers, B.V., 1993. To appear.
15. J. Smith. The identification of propositions and types in Martin-Löf's type theory: A programming example. In M. Karpinski, editor, *Foundations of Computation Theory,* volume 158 of *Lecture Notes in Computer Science,* pages 445–456, Berlin, 1983. Springer-Verlag.
16. D. Weise. Constraints, abstraction and verification. In M. Leeser and G. Brown, editors, *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects,* volume 408 of *Lecture Notes in Computer Science,* pages 25–39. Springer-Verlag, 1989.