HELMUT SCHWICHTENBERG AND KARL STROETMANN

FROM HIGHER ORDER TERMS TO CIRCUITS[1]

1. INTRODUCTION

In his lecture at the congress, the first author gave a survey on some recent results relevant for computability theory in the context of partial continuous functionals (cf. (Scott, 1982; Ershov, 1977; Stoltenberg-Hansen *et al.*, 1994)):

- An abstract definition of *totality* due to Berger (cf. (Berger, 1990; Berger, 1993) and (Stoltenberg-Hansen *et al.*, 1994, Ch. 8.3)), and applications concerning density and effective density theorems.
- Bounded fixed points: one can have the flexibility of fixed point definitions *and* termination at the same time (cf. (Schwichtenberg and Wainer, 1995)).
- A notion of *strict* functionals as a tool to prove termination of higher order rewrite systems (cf. (van de Pol and Schwichtenberg, 1995)).

Since this work is published already, we do not give details here but rather concentrate on another "applied" aspect of computability theory in higher types (also mentioned in the lecture): its possible use for the simultaneous design (from given components) and formal verification of hardware.

The basic observation is that many hardware units can be viewed as stream transformers, converting some input (control or data streams) into an output stream. This is possible even for bidirectional circuits since, in most cases, these can be modeled as pairs of unidirectional circuits. Here a stream is simply a function from the natural numbers (used to model time, i.e. the system clock) into the booleans (for control streams) or into some kind of data.

We consider some simple schemata (explicit definition and a form of primitive recursion) to define computable functionals. The resulting terms can be thought of as stream transformers, i.e. as circuits. The form of our schemata then makes it possible to directly translate a term into a circuit. On the other hand, we now have the term as a compact formal representation of the circuit, in the context of a reasonable theory, i.e. computability theory for partial continuous functionals. This is particularly useful for purposes of formal verification.

Our approach offers a number of benefits.

[1]The first author is partially supported by the working group NADA (New Hardware Design Methods) of the EC, and the second author is partially supported by a grant from the German Federal Ministry of Education, Science, Research, and Technology under contract number 01IS519A (project KORSYS).

1. It opens the possibility to treat some questions on hardware synthesis and verification in a "mathematically civilized" setting. After all, it is an old experience in computability theory that it pays if one does not unnecessarily restrict (higher order) arguments. Moreover, it provides the proper mathematical framework to deal with undefined or error objects.
2. Both the design and the verification of a circuit can be done in a modular way. In particular, the design and verification can start from components that are only specified abstractly by higher order formulas (Gordon, 1986).
3. The approach sketched can deal with various types of data abstractions. For example, in the example presented we will abstract completely from the bit level representation of the data.

It is these benefits that distinguish our approach from others that are based on representing circuitry by finite automata.

2. TERMS

We consider *recursive definitions* of stream transformers C in the form

$$C(\vec{a}, t) = M$$

with $\vec{a}$ a list of stream variables and M a term with free variables among $\vec{a}, t$ which is built inductively by means of the clauses

$$C(\vec{a}, \mathrm{pred}^k(t)) \quad \text{with } k \geq 1,$$
$$a_i(\mathrm{pred}^k(t)) \quad \text{with } k \geq 0,$$
$$D(M_1, \ldots, M_n).$$

Here we have written $\mathrm{pred}(t)$ for $t-1$ (which will be convenient in section 3); hence $\mathrm{pred}^k(t)$ is undefined if $t < k$. D ranges over already defined constants, and clearly a_i denotes the i–th component of the list $\vec{a}$. As an important special case we have *explicit definitions*, where M is built using the last two clauses only; this special case will correspond to combinatorial circuits in section 3. Among the constants we always have **if–then–else–fi**.

As an example we pick the minmax unit proposed in (Claesen, 1990). We have taken its formulation from the IFIP WG 10.2 collection of circuit verification examples[2]; cf. (Kropf, 1995). It is described there as the first non–trivial example which has gained some popularity and reveals some problems arising in the area of digital signal processor verification. We quote from the WWW page:

"The minmax unit has an input signal *in* which consists of a sequence of integers in the range of -256 to $+255$. The minmax unit has three boolean control signals *clear*, *reset* and *enable*. The unit produces an output sequence *out* at the same rate as *in* in the following way.

[2] These examples can be found at the URL `http://i81fs1.ira.uka.de/benchmarks/`.

1. *Out* is zero if *clear* is true, independent of the other control signals.
2. If *clear* is false and *enable* is false then *out* equals the last value of *in* before *enable* became false.
3. If *clear* is false and *enable* is true and *reset* is true then *out* follows *in*.
4. If *reset* becomes false, then *out* equals, on each time point t, the mean value of the maximum and minimum value of *in* until that time point."

Recall that we model input streams as functions from time (here: discrete time modeled by the natural numbers) to data (here: natural numbers). For the reasons of both simplicity and generality we take arbitrary natural numbers, not just those with a fixed bit length. We assume that we have units (ALUs) computing the sum, maximum and minimum of two numbers. These functions are understood in the strict sense, i.e. an error in any of the arguments produces an error in the value. We also need a unit for the (strict) function `half` defined by

$$\mathtt{half}(2n) = \mathtt{half}(2n+1) = n.$$

The specification talks about the "last value of *in* before *enable* became false". In order to design a unit yielding that value, consider the following recursive definition $(*)$:

$$\begin{aligned}
\mathtt{last}(\mathrm{in}, \mathrm{enable}, 0) := \ & \mathbf{if}\ \mathrm{enable}(0) \\
& \quad \mathbf{then}\ \mathrm{in}(0) \\
& \quad \mathbf{else}\ \mathrm{undef}_{\mathtt{nat}} \\
& \mathbf{fi}
\end{aligned}$$

$$\begin{aligned}
\mathtt{last}(\mathrm{in}, \mathrm{enable}, t+1) := \ & \mathbf{if}\ \mathrm{enable}(t+1) \\
& \quad \mathbf{then}\ \mathrm{in}(t+1) \\
& \quad \mathbf{else}\ \mathtt{last}(\mathrm{in}, \mathrm{enable}, t) \\
& \mathbf{fi}
\end{aligned}$$

Note that case distinction $0/t+1$ here is not really necessary; it also wrongly suggests that when describing a unit from these recursion equations we would need to perform a zero–test. We may write $(*)$ equivalently as $(**)$:

$$\begin{aligned}
\mathtt{last}(\mathrm{in}, \mathrm{enable}, t) := \ & \mathbf{if}\ \mathrm{enable}(t) \\
& \quad \mathbf{then}\ \mathrm{in}(t) \\
& \quad \mathbf{else}\ \mathtt{last}(\mathrm{in}, \mathrm{enable}, t-1) \\
& \mathbf{fi}
\end{aligned}$$

with the understanding that $0-1 = \mathrm{undef}_{\mathtt{nat}}$. (Hence (i) the final argument t of *last* ranges over the extended natural numbers now, (ii) we have to require $t \neq \mathrm{undef}_{\mathtt{nat}}$ in $(**)$ and (iii) we have to add $\mathtt{last}(\mathrm{in}, \mathrm{enable}, \mathrm{undef}_{\mathtt{nat}}) = \mathrm{undef}_{\mathtt{nat}}$).

Similarly we define the "the maximum and minimum value of *in* until that time point" (i.e. where *reset* becomes false) by

$$\begin{aligned}\mathtt{max}(\mathrm{in},\mathrm{reset},t) := \ & \textbf{if}\ \mathrm{reset}(t)\\ & \quad \textbf{then}\ \mathrm{in}(t)\\ & \quad \textbf{else}\ \mathtt{maximum}(\mathrm{in}(t),\mathtt{max}(\mathrm{in},\mathrm{reset},t-1))\\ & \textbf{fi}\end{aligned}$$

$$\begin{aligned}\mathtt{min}(\mathrm{in},\mathrm{reset},t) := \ & \textbf{if}\ \mathrm{reset}(t)\\ & \quad \textbf{then}\ \mathrm{in}(t)\\ & \quad \textbf{else}\ \mathtt{minimum}(\mathrm{in}(t),\mathtt{min}(\mathrm{in},\mathrm{reset},t-1))\\ & \textbf{fi}\end{aligned}$$

Now the minmax unit can be defined explicitly by

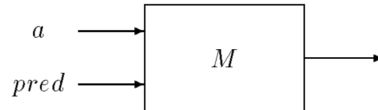
$$\begin{aligned}&\mathrm{out}(\mathrm{in},\mathrm{clear},\mathrm{reset},\mathrm{enable},t) \ =\\ &\quad \textbf{if}\ \mathrm{clear}(t)\\ &\quad\quad \textbf{then}\ 0\\ &\quad\quad \textbf{else if}\ \mathrm{enable}(t)\\ &\quad\quad\quad\quad \textbf{then}\ \mathtt{half}\big(\mathtt{max}(\mathrm{in},\mathrm{reset},t)+\mathtt{min}(\mathrm{in},\mathrm{reset},t)\big)\\ &\quad\quad\quad\quad \textbf{else}\ \mathtt{last}(\mathrm{in},\mathrm{enable},t)\\ &\quad\quad\quad \textbf{fi}\\ &\quad \textbf{fi}\end{aligned}$$

Note that among the constants for our example we have 0, +, **`maximum`**, **`minimum`**, and **`half`**.

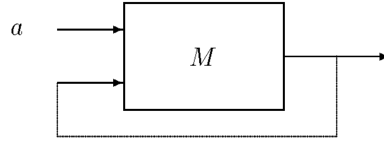
3. TRANSLATION INTO CIRCUITS

We begin with recursive definitions, i.e., $C(\vec{a},t) = M$ where M is a term with free variables among $\vec{a}, t$ built inductively by the three clauses above. We need a *register* unit

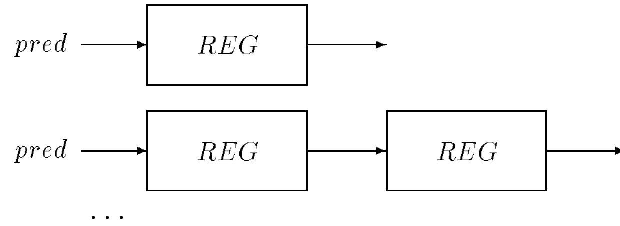
which at time $t+1$ gives out its input value at time t (and is undefined at time 0). For simplicity let us assume that $\vec{a} = a$. First for any such term $M[a,t]$ we inductively construct a circuit



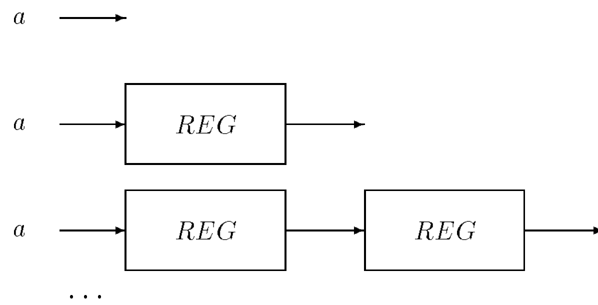
where both the `a` and the `pred` arrow may be missing. After this is done, we obtain the circuit for $C(a)$ by feedback:



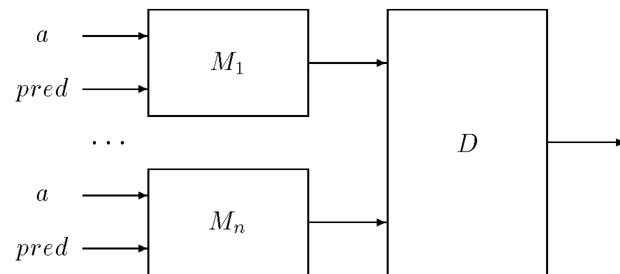
Case $C(a, \mathrm{pred}^k(t))$ with $k \geq 1$. For $k = 1, 2, \ldots$ take



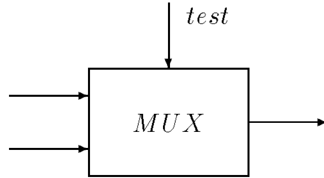
Case $a(\mathrm{pred}^k(t))$ with $k \geq 0$. For $k = 0, 1, 2, \ldots$ take



Case $D(M_1, \ldots, M_n)$.



As an example, let us construct circuits for the recursive definitions of `last`, `max` and `min`. Here we need a *multiplexer* unit



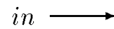
corresponding to **if**–**then**–**else**–**fi**. Recall the recursive definition of `last`:

$$\begin{aligned}\texttt{last}(\mathrm{in}, \mathrm{enable}, t) := \; & \mathbf{if}\; \mathrm{enable}(t) \\ & \quad \mathbf{then}\; \mathrm{in}(t) \\ & \quad \mathbf{else}\; \texttt{last}(\mathrm{in}, \mathrm{enable}, \mathrm{pred}(t)) \\ & \mathbf{fi}\end{aligned}$$

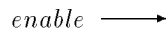
We now apply our inductive construction to the respective subterms of the right hand side.

$\texttt{last}(\mathrm{in}, \mathrm{enable}, \mathrm{pred}(t))$:

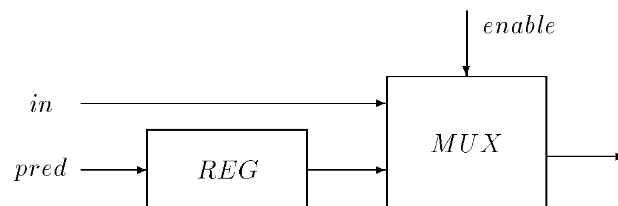
$\mathrm{in}(t)$:



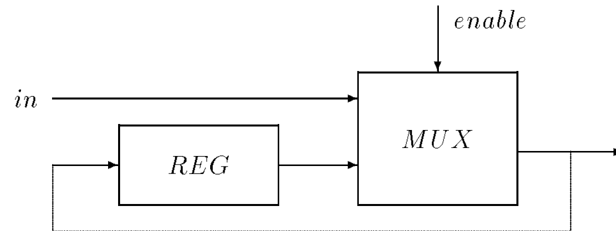
$\mathrm{enable}(t)$:



if $\mathrm{enable}(t)$ **then** $\mathrm{in}(t)$ **else** $\texttt{last}(\mathrm{in}, \mathrm{enable}, \mathrm{pred}(t))$ **fi**:



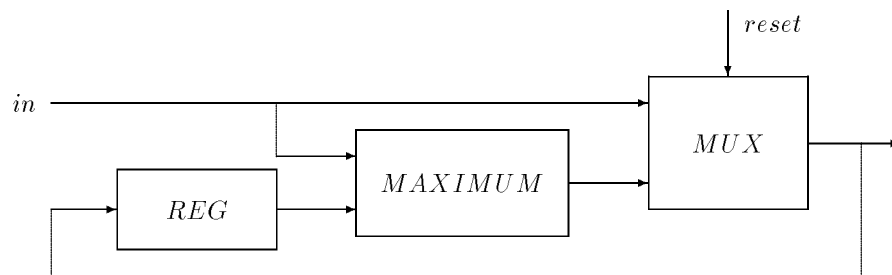
Finally our circuit for $\texttt{last}(\text{in}, \text{enable}, t)$ is built by feedback:



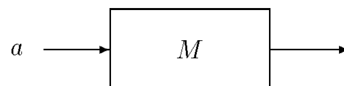
Similarly e.g. for

$$\begin{aligned}\texttt{max}(\text{in}, \text{reset}, t) := \ & \textbf{if}\ \text{reset}(t)\\ & \quad \textbf{then}\ \text{in}(t)\\ & \quad \textbf{else}\ \texttt{maximum}(\text{in}(t), \texttt{max}(\text{in}, \text{reset}, \text{pred}(t)))\\ & \textbf{fi}\end{aligned}$$

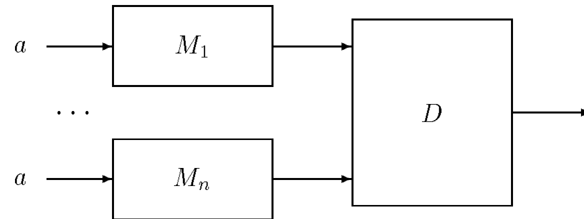
we obtain



For explicit definitions $C(a, t) = M[a, t]$ things are even simpler. For any such M we inductively construct a circuit



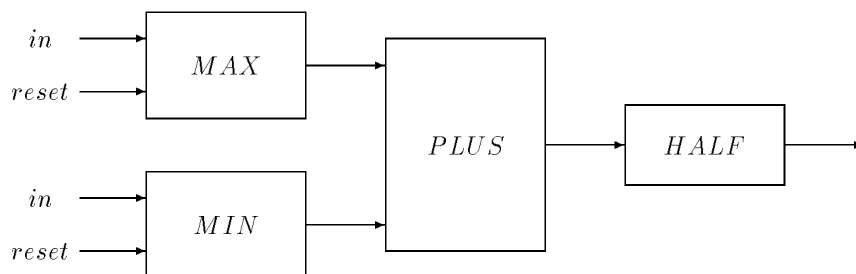
Case $a(\mathrm{t})$. Take $\mathtt{a}\longrightarrow$. *Case* $D(M_1, \ldots, M_n)$. Take



As an example, consider the explicit definition

$$\mathtt{mean}(\mathrm{in}, \mathrm{reset}, t) := \mathtt{half}(\mathtt{max}(\mathrm{in}, \mathrm{reset}, t) + \mathtt{min}(\mathrm{in}, \mathrm{reset}, t)).$$

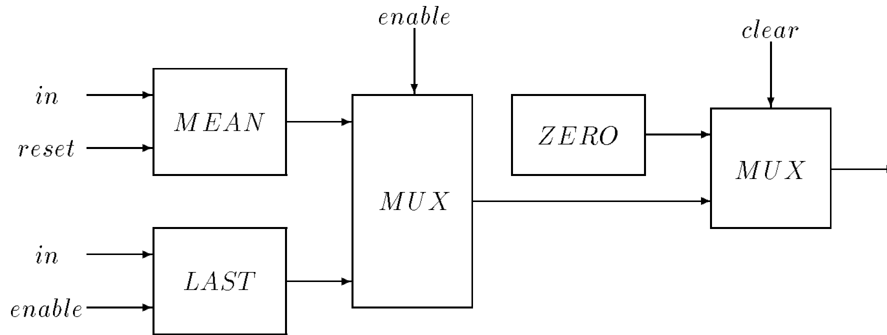
We construct a circuit to be called MEAN from MAX, MIN, PLUS and HALF by



Now we can transform the explicit definition

$$
\begin{array}{l}
\mathrm{out}(\mathrm{in}, \mathrm{clear}, \mathrm{reset}, \mathrm{enable}, t) \ = \\
\quad \textbf{if } \mathrm{clear}(t) \\
\quad\quad \textbf{then } 0 \\
\quad\quad \textbf{else if } \mathrm{enable}(t) \\
\quad\quad\quad\quad \textbf{then } \mathtt{half}\big(\mathtt{max}(\mathrm{in}, \mathrm{reset}, t) + \mathtt{min}(\mathrm{in}, \mathrm{reset}, t)\big) \\
\quad\quad\quad\quad \textbf{else } \mathtt{last}(\mathrm{in}, \mathrm{enable}, t) \\
\quad\quad\quad \textbf{fi} \\
\quad \textbf{fi}
\end{array}
$$

into a circuit:



4. SPECIFICATION

For readability we leave out the stream arguments and e.g. write out(t) for

$$\text{out}(\text{in}, \text{clear}, \text{reset}, \text{enable}, t).$$

We also omit leading universal quantifiers. The required properties 1–4 of the informal description of the minmax unit in section 2 then translate into

$$\begin{aligned} & \text{clear}(t) = \texttt{true} \\ \rightarrow\ & \text{out}(t) = 0, \end{aligned} \tag{1}$$

$$\begin{aligned} & \text{clear}(t) = \texttt{false} \wedge \text{enable}(t) = \texttt{false} \\ \rightarrow\ & \text{out}(t) = \texttt{last}(t), \end{aligned} \tag{2}$$

$$\begin{aligned} & \text{clear}(t) = \texttt{false} \wedge \text{enable}(t) = \texttt{true} \wedge \text{reset}(t) = \texttt{true} \\ \rightarrow\ & \text{out}(t) = \text{in}(t), \end{aligned} \tag{3}$$

$$\begin{aligned} & \text{clear}(t) = \texttt{false} \wedge \text{enable}(t) = \texttt{true} \wedge \text{reset}(t) = \texttt{false} \\ \rightarrow\ & \text{out}(t) = \texttt{half}(\texttt{max}(t) + \texttt{min}(t)). \end{aligned} \tag{4}$$

Note that we need non–strict equality here.

The first thing to observe is that the assumption on reset(t) in (4) is not necessary, i.e. we can prove the following strengthened form (4^*) of (4):

$$\begin{aligned} & \text{clear}(t) = \texttt{false} \wedge \text{enable}(t) = \texttt{true} \\ \rightarrow\ & \text{out}(t) = \texttt{half}(\texttt{max}(t) + \texttt{min}(t)). \end{aligned} \tag{4*}$$

However, this "strengthening" is slightly misleading, since we do need the assumption on reset(t) to prove that $\texttt{max}(t)$ and $\texttt{min}(t)$ have their expected properties, e.g. that $\texttt{max}(t)$ is the maximum value of *in* since *reset* became false. We split this up into two formulas: that $\texttt{max}(t)$ is an upper bound, and that it is the least upper bound. So we have to prove

$$\begin{aligned} & \mathrm{reset}(t) = \texttt{true} \\ \wedge\; & \big(\forall n\colon \texttt{nat}.\ n \neq 0 \wedge n \leq l \to \mathrm{reset}(t+n) = \texttt{false}\big) \\ \wedge\; & \big(\forall n\colon \texttt{nat}.\ n \leq l \to \mathrm{in}(t+n){\downarrow}\big) \\ \to\; & \forall n\colon \texttt{nat}.\ n \leq l \to \mathrm{in}(t+n) \leq \texttt{max}(t+l) \end{aligned} \tag{5}$$

$$\begin{aligned} & \mathrm{reset}(t) = \texttt{true} \\ \wedge\; & \big(\forall n\colon \texttt{nat}.\ n \neq 0 \wedge n \leq l \to \mathrm{reset}(t+n) = \texttt{false}\big) \\ \wedge\; & \big(\forall n\colon \texttt{nat}.\ n \leq l \to \mathrm{in}(t+n) \leq k\big) \\ \to\; & \texttt{max}(t+l) \leq k. \end{aligned} \tag{6}$$

Note that we have to require the definedness of $\mathrm{in}(t+n)$ in (5) (i.e. that $\mathrm{in}(t+n)$ is not $\mathrm{undef}_{\texttt{nat}}$); otherwise (5) would not hold, since $\mathrm{undef}_{\texttt{nat}} \leq \dots$ as well as $\dots \leq \mathrm{undef}_{\texttt{nat}}$ are defined to be false.

Finally, we have to formulate the specification of $\texttt{last}(t)$, i.e. that it is the "last value of *in* before *enable* became false":

$$\begin{aligned} & \mathrm{enable}(t) = \texttt{true} \\ \wedge\; & (\forall n\colon \texttt{nat}.\ n \neq 0 \wedge\ n \leq l \to \mathrm{enable}(t+n) = \texttt{false}) \\ \to\; & \texttt{last}(t+l) = \mathrm{in}(t). \end{aligned} \tag{7}$$

5. FORMAL VERIFICATION

It is now more or less routine to prove that the minmax circuit constructed in section 3 meets the specification given in section 4. However, we have not confined ourselves with a *paper and pencil* proof but rather have checked the correctness of our proof with machine help. In the presence of error values for input streams this seems to be particularly advisable, since it is easy for a human to forget some cases.

In fact, we have done the formal verification twice: First with the interactive prover MINLOG (Schwichtenberg, 1993) developed by the first author, and then again with the help of the automatic theorem prover SEDUCT (Stroetmann, 1995) under development at Siemens.

The MINLOG system is designed to deal with terms denoting computable functionals over the partial continuous functionals; hence the proof went rather smoothly. However, MINLOG is an interactive prover with only limited automated support (exept a mechanism to deal with equality logic which uses normalization of higher order terms (Berger and Schwichtenberg, 1991)). Therefore, it was a challenge to see to what extent a theorem prover with more automated components, but based on many sorted first order logic, could be used as well. It turned out that this was possible. We conclude with comments on some of the observations we made in the course of doing the proof.

The first problem is of course that the specification given in section 4 is written in *higher order logic* (Gordon, 1986), whilst SEDUCT is a theorem prover for many–sorted *first order logic*. We therefore have to translate this

specification into first order logic. However, the only part of this specification that is not first order is the use of the variables *in*, *clear*, *reset*, and *enable* as higher order variables, e.g. in $out(t)$. Now a closer inspection reveals that this use of higher order variables is not essential: we can eliminate expressions of the form $out(t)$ by introducing a new function symbol @ with signature

$$@ : \texttt{n_stream} \times \texttt{nat} \rightarrow \texttt{nat},$$

where `n_stream` is the type of *streams* of natural numbers. Of course, the intention is that for any stream of natural numbers f the value of $f\,@\,t$ is the same as $f(t)$. Since second order variables occur only in this context we can eliminate them by replacing all expressions of the form $f(t)$ where f is a second order variable with the expression $f\,@\,t$.

A point of general interest that we learned while carrying out the proof is the following. If the formulas to be proved contain quantified subformulas, then it seems to be a good idea to define predicates equivalent to these subformulas. For example, property (6) contains the universally quantified subformula

$$\forall n\colon \texttt{nat}.\ n \leq l \rightarrow in\,@\,(t+n) \leq k.$$

To eliminate this subformula we have introduced the predicate `bounded` satisfying an appropriate lemma. This lemma could then be used to eliminate the above subformula from (6).

ACKNOWLEDGEMENTS

The authors would like to acknowledge the fact that the paper has benefited from a number of comments made by Dr. Gerd Venzl.

REFERENCES

Ulrich Berger. *Totale Objekte und Mengen in der Bereichstheorie.* PhD thesis, Mathematisches Institut der Universität München, 1990.

Ulrich Berger. Total sets and objects in domain theory. *Annals of Pure and Applied Logic*, 60:91–117, 1993.

Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In Rao Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, Los Alamitos, 1991.

Luc Claesen, editor. *Formal VLSI Specification and Synthesis VLSI Design Methods I and II.* IFIP, sponsored by IMEC, North–Holland, Amsterdam, 1990.

Yuri L. Ershov. Model C of partial continuous functionals. In R. Gandy and M. Hyland, editors, *Logic Colloquium 1976*, pages 455–467. North–Holland, Amsterdam, 1977.

Mike Gordon. Why higher–order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. Elsevier, Amsterdam, 1986.

Thomas Kropf. Benchmark–circuits for hardware–verification. In R. Kumar and T. Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 901 of *Lecture Notes in Computer Science*, pages 1–12. Springer Verlag, Berlin, Heidelberg, New York, 1995.

Jaco van de Pol and Helmut Schwichtenberg. Strict functionals for termination proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 350–364. Springer Verlag, Berlin, Heidelberg, New York, 1995.

Helmut Schwichtenberg. Proofs as programs. In P. Aczel, H. Simmons, and S.S. Wainer, editors, *Proof Theory. A selection of papers from the Leeds Proof Theory Programme 1990*, pages 81–113. Cambridge University Press, 1993.

Helmut Schwichtenberg and Stanley S. Wainer. Ordinal bounds for programs. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 387–406. Birkhäuser, Boston, 1995.

Dana Scott. Domains for denotational semantics. In E. Nielsen and E.M. Schmidt, editors, *Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer Verlag, Berlin, Heidelberg, New York, 1982. A corrected and expanded version of a paper prepared for ICALP'82, Aarhus, Denmark.

Viggo Stoltenberg-Hansen, Edward Griffor, and Ingrid Lindström. *Mathematical Theory of Domains*. Cambridge tracts in Theoretical Computer Science. Cambridge University Press, 1994.

Karl Stroetmann. SEDUCT — a proof compiler for first order logic. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software. Final Report*, volume 1009 of *Lecture Notes in Computer Science*, pages 299–316. Springer Verlag, Berlin, Heidelberg, New York, 1995.

Helmut Schwichtenberg
Mathematisches Institut der Universität München
Theresienstr. 39, D-80333 München

email: `schwicht@rz.mathematik.uni-muenchen.de`

Karl Stroetmann
Siemens AG, Corporate Research and Development
Otto–Hahn–Ring 6, D–81739 München

email: `Karl.Stroetmann@zfe.siemens.de`