NADA-Workshop on Streams, Oct 16-18, 1995
Schloß Elmau, Klais, Germany

Organizers: H.Leiß, H.Schwichtenberg

1 Abstracts

Infinite objects and infinite proofs

Thorsten Altenkirch, LFCS, University of Edinburgh
email: alti@dcs.ed.ac.uk

We view Type Theory as a functional programming language which

- is total,
- has dependent types.

This sort of language is interesting because it combines a proof system and a programming language.

How do we achieve totality? Following a suggestion by T.Coquand [Coq92], we use a syntactic (decidable) criteria to identify *structural recursion*. A functional program is structural recursive, if the structural ordering (or a lexical extension of it) of an inductively defined datatype is always reduced during recursion. The structural ordering on a datatype is the subterm ordering on canonical terms which includes instances of functions in the case of higher order inductive definitions.

Using structural recursion all the standard elimination constants of Type Theory (as proposed by Martin-Löf) can be defined. However, in many cases structural recursion leads to much simpler proofs and to more realistic programs (e.g. in terms of efficiency). This is especially important when using a computer implementation of Type Theory, such as the ALF system [AGNvS94, MN94].

We can derive a general principle which allows us to implement any function which uses *provable well-founded recursion* from structural recursion (this is based on ideas in [Nor88, Pau86]). Using structural recursion may be less efficient since we have to calculate the bounds of recursions explicitly. However, it seems that these useless computations can be eliminated by syntactic optimizations.

In [Coq94] T.Coquand proposes to extend this approach to *infinite objects* like streams (They are often called *coinductive types* because they arise as they categorical dual of the usual inductively defined datatypes). Instead of termination we are interested in *productivity*, which means that a term should have a head normal form and this property is inherited by subterms. Coquand also proposes a notion dual to structural recursion: *guarded recursion*. Essentially, this means that every recursive call has to be guarded by a constructor (this is reminiscent of the notion of guardedness in process algebras). Coquand shows that guardedness implies productivity.

It turns out that we can also derive a *coinduction* principle from guarded recursion. By coinduction we mean that every relation which is preserved by all constructors is already included in the identity. However, in intensional Type Theory we can only derive coinduction upto a congruence which is preserved by all definable operations.

Not all recursive programs which are productive arise from guarded recursion. Counterexamples are already the definition of Hamming streams and the stream of prime numbers based on the sieve of Eratosthenes. Hence we are looking for a principle which dualizes wellfounded recursion.

For the special cases of streams we propose a condition on stream processors which we call liveness (see also [Hof95], pp.234). A function $f \in \mathit{Stream} \to \mathit{Stream}$ is live if it produces a constant head a and $\mathit{tail} \circ f \circ (a :: \_)$ is *live* (note that this is a coinductive definition). We can show that every live function has a unique fixpoint.

Our approach to infinite objects and computations can be summarized by the following table:

wellfounded trees	infinite trees
terminating	productive
structural recursion	guarded recursion
induction	coinduction
wellfounded recursion	liveness (?)

We are interested to derive a general principle for productive recursions. Is guarded recursion sufficient to construct productive programs? And can we optimize guarded recursive programs as we did for structural recursion?

References

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A User's Guide to ALF*. Chalmers University of Technology, Sweden, May 1994. Available on the WWW `file://ftp.cs.chalmers.se/pub/users/alti/alf.ps.Z`.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[Coq94] Thierry Coquand. Infinite Objects in Type Theory. In *Types for Proofs and Programs*, LNCS, pages 62–78, Nijmegen, 1994. Springer-Verlag.

[Hof95] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, University of Edinburgh, 1995.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[Nor88] Bengt Nordström. Terminating General Recursion. *BIT*, 28(3):605–619, October 1988.

[Pau86] Lawrence C. Paulson. Constructing Recursion Operators in Intuitionistic Type Theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

Experiments with the Alternating Bit Protocol: Verification and Program Extraction

Ulrich Berger and Monika Seisenberger, Mathematisches Institut, Universität München

We have verified a generalization of the Alternating Bit Protocol (ABP) and extracted from a part of the correctness proof (which uses classical logic) a program estimating the delay of transmission. The formalization and program extraction have been carried out in the proof system MINLOG [1].

In the version of the ABP we considered, the channels may loose, delay and dublicate data (usually they may just loose data, and whenever a datum is transmitted then only once and without delay). As a formal system we use first order predicate logic with function variables. Streams are modelled by functions from discrete time to data and channels by functions from streams to streams. We chose this representation in order to make the specification and verification immediately understandable without requiring knowledge obout special calculi like temporal logic [5], process algebra [3] or formal languages [6]. It has turned out that the verification whas quite manageable even though in our approach we had to talk explicitly about time and data, and quantified over them (which is avoided by other approaches).

One part in the verification of the ABP consists in proving that the protocol doesn't get stuck, i.e., every input datum is transmitted eventually. A somewhat weaker statement is that for every time n there exists a time k such that the datum arrived at k has been sent after n. Let $r(k)$ denote the time when the datum was sent which arrived at time k. Then the above statement reads formally

$$\forall n \exists k \, (n \leq r(k)). \tag{1}$$

The number $k - n$ can be considered as an estimate of the time one has to wait until some datum sent after n will be transmitted. The formula above follows easily from

$$\forall k_1 \exists k_2 \geq k_1 \, (r(k_1) < r(k_2)). \tag{2}$$

Given functions estimating the delay caused by the faulty but fair channels one can prove (2) by contradiction: Given k_1, from the assumption $\forall k_2 \geq k_1 \, (r(k_1) \not< r(k_2))$ a contradiction is derived. It is well known that such classical (non constructive) existence proofs always can be transformed into constructive ones. An easy method for doing this is provided by H. Friedman's A–translation [4], which is discussed in [1] and refined in [2]. We implemented the classical proof of (1) in MINLOG and obtained via program extraction a nice algorithm estimating the delay. From this algorithm one can easily read off qualitative properties of the ABP. E.g.,if the channels have constant respectively linear delay then the ABP has linear respectively exponential delay.

References

[1] Ulrich Berger and Helmut Schwichtenberg. Program development by proof transformation. In *Proceedings of the NATO Advanced Study Institute, International Summer School* Marktoberdorf, Germany, (1993) 299–340

[2] Ulrich Berger and Helmut Schwichtenberg: Program Extraction from Classical Proofs. Logic and Computational Complexity, International Workshop LCC '94, Indianapolis, IN, USA, October 1994. Daniel Leivant, editor. Lecture Notes in Computer Science, Vol. 960 (1995) 77–97

[3] Marc Bezem and Jan Friso Groote. A Formal Verification of the Alternating Bit Protocol in the Calculus of Constructions. Department of Philosophy, Utrecht University. Logic Group Preprint Series 88 (1993)

[4] Harvey Friedman. Classically and intuitionistically provably recursive functions. In Dana S. Scott and Gert H. Müller, *Higher Set Theory*, Springer Lecture Notes in Mathematics 699 (1978)

[5] Fred Kröger. Temporal Logic of Programs. EATCS Monographs on Theoretical Computer Science, Vol 8. Springer (1987)

[6] Bernhard Möller. Ideal Streams. Proc. IFIP TC2 Conf. on Programming Concepts, Methods and Calculi, San Miniato (1993)

Timed Streams and Abstractions

Manfred Broy, Institut für Informatik, Technische Universität München, Germany

We discuss a method for the specification of reactive asynchronous components with a concurrent access interface and outline its mathematical foundation. The method supports the specification of components that show a complex reactive behavior including timing aspects. Examples are the nonstrict fair merge or the arbiter. We show that approximation and timing are closely related. The method supports the specification of reactive systems and their modular composition into data flow networks. The specification approach is compositional. It supports the integrated specification and verification of both safety and liveness conditions in modular system descriptions.

Calculating Hamming Streams

Walter Dosch, Institut für Mathematik, Universität Augsburg
email: dosch@uni-augsburg.de

1.1 Introduction

This paper concentrates on methods for the deductive design of stream processing functions. As a case study we treat the problem of computing the infinite stream of Hamming numbers $2^i \cdot 3^j \cdot 5^k$ $(i, j, k \in \mathcal{N})$ in strictly ascending order.

In [Dijkstra 76] the problem is attributed to R. Hamming. Dijkstra develops an imperative solution for a finite sequence of Hamming numbers avoiding the multiple computation of the same element. Further imperative solutions are verified in [Gries 81] and [Dijkstra, Feijen 85]. [Partsch 90] gives a transformational development passing from a prealgorithmic specification via a functional solution to the imperative program.

[Wadge, Ashcroft 85] present a dataflow network in Lucid computing the infinite stream of Hamming numbers. Recently, [Meinke, Steggles 95] give a non-constructive higher-order equational Horn specification and verify the correctness of the dataflow algorithm.

1.2 The Set of Hamming Numbers

As a major design decision, we structure the problem solution into two subtasks. In this section we calculate the *set* of Hamming numbers, in the next section the *stream* of Hamming numbers.

The infinite set $h \subseteq \mathcal{N}$ of Hamming numbers is specified by the equation

$$h \quad = \quad 2^{\mathcal{N}} \cdot 3^{\mathcal{N}} \cdot 5^{\mathcal{N}} . \tag{3}$$

For notational convenience, we extend all functions elementwise to sets. Moreover, we will drop brackets around singleton sets and assume that arithmetic operators bind tighter than set operators.

We abstract from this particular example by generalizing the function used in the specification (3). So we compute the set

$$H = f(\mathcal{N}, \mathcal{N}, \mathcal{N}) \tag{4}$$

of natural numbers given by a ternary function $f\colon \mathcal{N} \times \mathcal{N} \times \mathcal{N} \to \mathcal{N}$. We require that the function f follows in each argument a pattern of primitive recursion

$$f(x+1, y, z) \quad = \quad f_1(f(x, y, z)) \tag{5}$$

$$
\begin{aligned}
f(x, y+1, z) &= f_2(f(x,y,z)) && (6)\\
f(x, y, z+1) &= f_3(f(x,y,z)) && (7)
\end{aligned}
$$

with strictly monotonic and properly increasing functions f_i: $\mathcal{N} \to \mathcal{N}$ $(i \in \{1,2,3\})$:

$$
\begin{aligned}
x < y &\Rightarrow f_i(x) < f_i(y) && (8)\\
x \geq f(0,0,0) &\Rightarrow x < f_i(x) && (9)
\end{aligned}
$$

With the instantiation $f_1(x) = 2 \cdot x,\ f_2(x) = 3 \cdot x,\ f_3(x) = 5 \cdot x$, and $f(x,y,z) = 2^x \cdot 3^y \cdot 5^z$ we retrieve Hamming's problem.

We aim at deriving a recursive solution for H by imposing an induction principle. Starting from the simple induction $\mathcal{N} = 0 \cup \mathcal{N}{+}1$ on natural numbers, we calculate for triples of natural numbers:

$$\mathcal{N}{\times}\mathcal{N}{\times}\mathcal{N} \quad = \quad 0{\times}0{\times}0 \cup (\mathcal{N}{+}1){\times}\mathcal{N}{\times}\mathcal{N} \cup \mathcal{N}{\times}(\mathcal{N}{+}1){\times}\mathcal{N} \cup (\mathcal{N}{+}1){\times}\mathcal{N}{\times}\mathcal{N} \qquad (10)$$

Using the distributivity of function application over set union and the properties of the recursion scheme (5)–(7), we derive from (4) the equation

$$H \quad = \quad f_0 \cup f_1(H) \cup f_2(H) \cup f_3(H)\,, \qquad (11)$$

where f_0 abbreviates $f(0,0,0)$. This fixpoint equation has a unique solution; Kleene's approximation gives an algorithm for computing the infinite set.

With the instantiation above, we obtain the set of Hamming numbers

$$h \quad = \quad 1 \cup 2{\cdot}h \cup 3{\cdot}h \cup 5{\cdot}h\,; \qquad (12)$$

this completes the first subtask.

1.3 The Stream of Hamming Numbers

We consider the domain

$$A^\infty \quad = \quad A^\star \cup A^\omega \qquad (13)$$

of finite and infinite, strictly *a*scending sequences of natural numbers partially ordered by the prefix relation.

The stream $\vec{S} \in A^\omega$ of Hamming numbers is obtained by ordering the set H. The implementation step from sets to ordered sequences is described using an abstraction function *set*: $A^\infty \to \wp(\mathcal{N})$. On finite streams it can be specified inductively by

$$
\begin{aligned}
set(\langle\rangle) &= \emptyset && (14)\\
set(x \lhd \vec{x}) &= x \cup set(\vec{x}) \qquad (x \lhd \vec{x} \in A^\star)\,. && (15)
\end{aligned}
$$

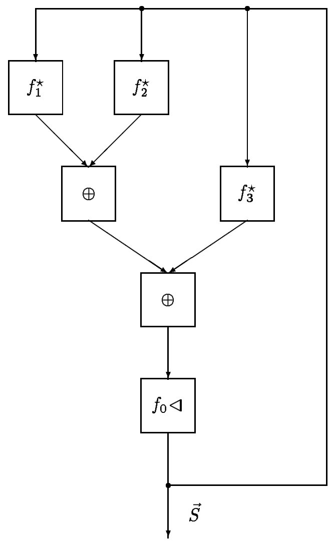
By monotonicity, it has a unique continuous extension to infinite streams.

For the pointwise application of a function to all elements of a stream, we use the apply-to-all functional $.^\star$: $[\mathcal{N} \xrightarrow{\text{strictly monotonic}} \mathcal{N}] \to [A^\infty \to A^\infty]$. On finite streams it is given by

$$
\begin{aligned}
h^\star(\langle\rangle) &= \langle\rangle && (16)\\
h^\star(x \lhd \vec{x}) &= h(x) \lhd h^\star(\vec{x})\,. && (17)
\end{aligned}
$$

The set abstraction and the map functional commute:

$$set \circ h^\star \quad = \quad h \circ set \qquad (18)$$



The stream $\vec{S} \in A^\omega$ of Hamming numbers is implicitly specified by

$$set(\vec{S}) \quad = \quad H\,. \qquad (19)$$

Now it is straightforward to derive a recursive solution:

$$
\begin{array}{lll}
 & set(\vec{S}) & \\
= & H & \{\text{unfold } set(\vec{S})\ (19)\}\\
= & f_0 \cup f_1(H) \cup f_2(H) \cup f_3(H) & \{\text{unfold } H\ (11)\}\\
= & f_0 \cup f_1(set(\vec{S})) \cup f_2(set(\vec{S})) \cup f_3(set(\vec{S})) & \{\text{fold } set(\vec{S})\ (19)\}\\
= & f_0 \cup set(f_1^\star(\vec{S})) \cup set(f_2^\star(\vec{S})) \cup set(f_3^\star(\vec{S})) & \{\text{use } (18)\}\\
= & f_0 \cup set((f_1^\star(\vec{S}) \oplus f_2^\star(\vec{S})) \oplus f_3^\star(\vec{S})) & \{\text{introduce } \oplus\ (20)\}\\
= & set(f_0 \lhd ((f_1^\star(\vec{S}) \oplus f_2^\star(\vec{S})) \oplus f_3^\star(\vec{S}))) & \{\text{fold } set\ (15) \text{ using } (8) \text{ and } (9)\}
\end{array}
$$

where the auxiliary function $\oplus$: $A^\omega \times A^\omega \to A^\omega$ is specified by

$$set(\vec{x} \oplus \vec{y}) \quad = \quad set(\vec{x}) \cup set(\vec{y}) \quad (\vec{x}, \vec{y} \in A^\omega)\,. \qquad (20)$$

This equation is established by the fixpoint equation

$$\vec{S} \quad = \quad f_0 \lhd ((f_1^\star(\vec{S}) \oplus f_2^\star(\vec{S})) \oplus f_3^\star(\vec{S})) \qquad (21)$$

which has again a unique solution. It describes the above network of computing agents. With the specialization

$$\vec{s} \quad = \quad 1 \lhd (((2\cdot)^\star(\vec{s}) \oplus (3\cdot)^\star(\vec{s})) \oplus (5\cdot)^\star(\vec{s})) \qquad (22)$$

we solve Hamming's problem.

1.4 The Ordered Merge Function

It remains to develop the *ordered merge function* $\oplus$: $A^\infty \times A^\infty \to A^\infty$ which transfers the set union to the stream level.

We aim at implementing it by repeatedly comparing the first elements of the input streams. The ordered merge can only progress if for each element of one input stream there occurs a comparison element within the other input stream. Hence we introduce the interval $[\![.]\!]$: $A^\infty \to \wp(\mathcal{N})$ represented by a stream. For finite streams it can be defined inductively by

$$[\![\langle\rangle]\!] = \emptyset \tag{23}$$

$$[\![x \lhd \vec{x}]\!] = [0..x] \cup [\![\vec{x}]\!]\,. \tag{24}$$

The interval function is monotonic and allows a continuous extension to infinite streams. Moreover we have $[\![\vec{x}]\!] = \mathcal{N}$ iff $\vec{x} \in A^\omega$. The ordered merge function $\oplus$: $A^\star \times A^\star \to A^\star$ on finite streams is then specified by

$$set(\vec{x} \oplus \vec{y}) = (set(\vec{x}) \cap [\![\vec{y}]\!]) \cup ([\![\vec{x}]\!] \cap set(\vec{y})) \qquad (\vec{x}, \vec{y} \in A^\star)\,. \tag{25}$$

¿From this equation we derive the following recursive solution for the finite behaviour:

$$\vec{x} \oplus \langle\rangle = \langle\rangle \tag{26}$$

$$\langle\rangle \oplus \vec{y} = \langle\rangle \tag{27}$$

$$(x \lhd \vec{x}) \oplus (y \lhd \vec{y}) = \begin{cases} x \lhd (\vec{x} \oplus (y \lhd \vec{y})) & \text{if } x < y \\ x \lhd (\vec{x} \oplus \vec{y}) & \text{if } x = y \\ y \lhd ((x \lhd \vec{x}) \oplus \vec{y}) & \text{if } x > y \end{cases} \tag{28}$$

The ordered merge function on finite streams is well-defined, commutative, associative, idempotent and strict in both arguments. Moreover, it is monotonic and can naturally be extended to a continuous merge function on infinite streams which — by construction — validates the specification (20).

1.5 Conclusion

This case study shows that deductive design needs adequate *abstractions* and suitable *structuring*. When developing stream functions, we need flexible methods to relate the *finite* and the *infinite behaviour*. A good design explains the *algorithmic principles* and allows the *systematic redesign* of variants.

References

E.W. Dijkstra: *A Discipline of Programming.* Englewood Cliffs, N.J.: Prentice-Hall 1976

E.W. Dijkstra, W.H.J. Feijen: *Methodik des Programmierens.* Bonn: Addison-Wesley 1985

D. Gries: *The Science of Programming.* Berlin: Springer 1991

K. Meinke, L.J. Steggles: *Correctness of Dataflow and Systolic Algorithms: Case Studies in Higher-Order Algebra.* Manuscript, Department of Computer Science, University College of Swansea, 1995

H.A. Partsch: *The Specification and Transformation of Programs — A Formal Approach to Software Development.* Berlin: Springer 1990

W. Wadge, A. Ashcroft: *Lucid — The Dataflow Programming Language.* New York: Academic Press 1985

From terms to circuits: the minmax example

Helmut Schwichtenberg, Mathematisches Institut der Universität München, Germany

A hardware unit can be viewed as a stream transformer, converting some input (control or data streams) into an output stream. This functional view seems to be adequate (cf. Fourman [2]); in particular, bidirectional transistors are rarely used in practice. Moreover, it makes the mathematical modelling of hardware units straightforward: Any unit (ALU or sequential) can de viewed as a higher order term. One additional benefit is that one can test a unit without building it: First form the closed application term starting with the term representing the unit followed by terms representing concrete input streams and a concrete point in time. Then evaluate the resulting closed term of ground type. Of course, one can always describe functionals using relations (as done in HOL by Gordon [3, 4]), but for the reasons given we prefer the functional approach.

By a stream we simply mean a function from the natural numbers (used to model time, i.e. the system clock) into the booleans (for control streams) or data. We also allow an explicit error object in each data type, denoted e.g. by $\bot_{nat}$ or $\bot_{boole}$. Then a hardware unit is a computable continuous functional of type level two; cf. Stoltenberg–Hansen's book [7] for the mathematical theory involved.

Here we do not need much theory, but rather want to demonstrate with a simple example how — starting from an informal description — one can simultaneously

- design (from given components) and
- formally verify

a circuit. The example we pick is the minmax unit proposed by Luc Claesen [1]. See Section 2, Examples.

References

[1] L. Claesen, editor. *Formal VLSI Specification and Synthesis VLSI Design Methods I and II.* IFIP, sponsored by IMEC, North–Holland, 1990.

[2] M. Fourman, W. Palmer, and R. Zimmer. Using higher–order functions to describe hardware, 1989.

[3] M. Gordon. Why higher–order logic is a good formalism for specifying and verifying hardware, 1986.

[4] M. Gordon. A proof generating system for higher–order logic. Technical Report 103, University of Cambridge, Computer Laboratory, January 1987.

[5] T. 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, 1995.

[6] L. Pierre. The formal proof of the 'min–max' sequential benchmark decribed in CASCADE using the Boyer–Moore theorem prover. In L. Claesen, editor, *Formal VLSI Specification and Synthesis VLSI Design Methods II*, pages 329–348. IFIP, sponsored by IMEC, North–Holland, 1990.

[7] V. Stoltenberg-Hansen, E. Griffor, and I. Lindström. *Mathematical Theory of Domains.* Cambridge tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1994.

Stream models?

Viggo Stoltenberg-Hansen, Department of Mathematics, Uppsala University, Sweden

Models of streams and of operations on streams are surveyed with the intention of initiating discussion and clarification of the usefulness of various models for hardware.

A Simple Characterization of the Continuous Functionals

Viggo Stoltenberg-Hansen, Department of Mathematics, Uppsala University, Sweden
(Report on joint work, in progress, with Dag Normann and Erik Palmgren)

Many equivalent characterisations of the Kleene-Kreisel continuous functionals have been given. Perhaps the most elegant is the one due to Ershov using domain theory, where the continuous functionals are extracted as the total elements of the partial continuous functionals (up to a natural equivalence). We give a finitary characterisation using a Frechèt product of the type structures built up over the finite initial segments of the base type N. A natural symmetric and transitive relation is defined on the product. The total elements of a given type in the product are defined to be those which are equivalent to themselves. It is shown that the total elements (up to the equivalence) are naturally isomorphic to the continuous functionals via Ershov's characterisation. The method has been extended to transfinite type structures with dependent types and empty types. It is conjectured that the method yields a model for Martin-Löf type theory.

Refining Ideal Behaviours

Bernhard Möller, Institut für Mathematik, Universität Augsburg, Germany
email: moeller@uni-augsburg.de

We provide some mathematical properties of *behaviours* of systems, where the individual elements of a behaviour are modeled by *ideals* of a suitable partial order. It is well-known that the associated ideal completion provides a simple way of constructing algebraic cpos. An ideal can be viewed as a set of consistent finite or compact approximations of an object which itself may even be infinite.

We introduce a special way of characterising behaviours through sets of relevant approximations. This is a generalisation of the technique used in [1] for the case of streams. Given a set $P \subseteq M$ of a partial order $(M, \leq)$, we define

$$\mathsf{ide}\, P := \{Q^{\leq} : Q \subseteq P \text{ directed}\} ,$$

where $Q^{\leq} := \{x \in M : \exists y \in Q : x \leq y\}$ is the downward closure of Q. So $\mathsf{ide}\, P$ is the set of all ideals "spanned" by directed subsets of P. We prove a number of distributivity and monotonicity laws for ide and related operators. They are the basis for correct refinement of specifications into implementations. They are used in [2] in the algebraic calculation of a bounded queue module.

References

[1] B. Möller: Ideal streams. In: E.-R. Olderog (ed.): Programming concepts, methods and calculi. IFIP Transactions A-56. Amsterdam: North-Holland 1994, 39–58

[2] B. Möller: Calculating a bounded queue. Extended version of Document 734 HKO-7 of IFIP WG2.1, Hong Kong, Jan. 1995 (forthcoming)

[3] B. Möller: Refining ideal behaviours. Institut für Mathematik der Universität Augsburg, Report Nr. 345, 1995

Streams, States and Refinement

Carlos Delgado Kloos, Peter T. Breuer, Universidad Politecnica de Madrid

Contents: Signals and Causal Continuous Systems
Timed Streams and Causal Discrete Time-Preserving Systems
Streams and Causal Discrete Untimed Systems
States and State Machines
Refinement High-Level Refinement (From Predicates to Functions)
Low-Level Refinement (From Functions to Code)

We defined signals and several kinds of streams, and stated what causality means for each. We chose to concentrate on streams as finite or infinite lists. We defined the derivative of a stream transformer and showed how to generate a state machine from it. The interesting cases are when the machine is finite, because then the stream transformer can be implemented as hardware.

We went on to look at refinement of stream transformer specifications via the state machine derivative. We concluded that refinement of a predicate to a functional stream transformer was not going to work this way, but that we could do refinement of stream transformer functionalities to imperative code. We gave a refinement principle and showed how it could be used to generate imperative code for an oscillator from a statement of the stream-to-stream functionality alone.

2 Examples

The Case of the Differental Amplifier

Carlos Delgado Kloos, Universidad Politécnica de Madrid
email: cdk@dit.upm.es

Consider a simple operational amplifier that performs the function g:

$$g(x, y) = D * (x - y)$$

where D is the gain of the amplifier. Now we feed a copy of the output back to the second input. The associated function f could be described as:

$$f(x) = D * (x - f(x))$$

One now might interpret this equation in different settings:

- As a function definition over the reals in a programming language, this defines a divergent function.
- One might build this as an analog electronic system and simply solve the equation $y = D * (x - y)$ eliminating y. One gets then $y = (D/(D + 1)) * x$, which makes sense for $D \neq -1$.
- One might view this as system evolving through time and consider a little delay at the feedback path. This delay is very small in comparison with a rate of change at the input. Under this view, the outputs form a geometric series that converges with $|D| < 1$ to $(D/(D + 1)) * x$.
- Finally, one might build the mechanical system corresponding to the equation. It turns out that the result is the same, but with a different convergence condition: $D \neq 1$.

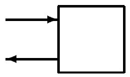
The questions are now: How are these different interpretations related? Why are the results equal, but the convergence conditions different? How is discrete modelling related to continuous modelling?

This interesting case was originally pointed out to me by Raymond Boute.

A Generalized Priority Queue (GPQ)

Bernhard Möller, Institut für Mathematik, Universität Augsburg, Germany

The GPQ module has an input and an output line. We first assume unbounded capacity.



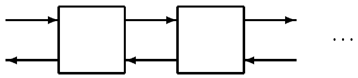
It may receive data from some set D. Upon an input request at time t, it assigns to the datum x it receives a priority $p(t,x)$ according to a function $p : T \times D \to P$, where T is the time domain and P is some linearly ordered set of priorities. Upon an output request it emits the datum which currently has the highest priority, among those with equal priority the one that was input first.

Specializations:

- $p(t,x)$ independent of t: usual priority queue.
- p constant: usual FIFO queue.
- $p(t,x) = t$: LIFO stack.

Tasks:

- Specify this formally.
- Characterise sequences of in/output requests that guarantee eventual output of every datum.
- Alternatively, find restrictions on p that achieve this.
- The above description assumes unbounded capacity. Discuss bounded variants and systematic transformation from unbounded to bounded.
- Derive a schematic implementation of the following shape (unbounded case) by deductive design and adapt it to the above special cases and/or boundedness:



Every cell is supposed to hold one datum. The leftmost cell communicates with the environment. It holds the current datum with the highest priority. Input of a datum is realized by having the datum trickle to the right till it has found its place; then it is inserted into the respective cell while the remaining contents of the queue is shifted to the right. Output consists in emitting the datum in the leftmost cell and shifting all other cell contents to the left.

C. Delgado Kloos has pointed out that GPQ can be reduced to the usual priority queue (UPQ) by prefixing UPQ with a module that adds time stamps to priorities and postfixing by another that removes them. Describe this formally.

Min-Max Unit

Helmut Schwichtenberg, Mathematisches Institut der Universität München, Germany

This example has been proposed by Luc Claesen [1]. We have taken its formulation from the IFIP WG 10.2 collection of circuit verification examples (`http://goethe.ira.uka.de/benchmarks/`); cf. [2]. 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. This example was used at the international workshop on “Applied Formal Methods For Correct VLSI Design” and several solutions can be found in the proceedings [1], in particular one by Laurence Pierre [3].

Informal description of the minmax unit

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.

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.”

It is also stated in the WWW page that “the above specification is not as unambiguous as it seems at the first glance. To point this out, some comments of G. Janssen from Eindhoven University are added below.” The comments are

1. Is this independent of *reset*? (Answer: yes). What is if *clear* is false and *enable* is false from start? (Answer here: then *out* is $\perp_{nat}$ and keeps that value till some other control combination takes over).
3. What does “follow” exactly mean? (Answer: “follow” means equals).
4. (a) What must be the values of *clear* and *enable*? (Answer: *clear* must be false and enable must be true). (b) Does “until” include or exclude the current time point? (Answer: “until” is interpreted to mean inclusive). (c) What if *reset* is false from start? (Answer here: then min and max are set to $\perp_{nat}$). (d) When are the minimum and maximum values (re)initialized and to what value? Where does the averaging start? (Answer: Only when *reset* makes a transition from true to false are min and max reinitialized. At the time *reset* is false, min and max are set to *in*; averaging starts from afresh). (e) What if *clear* and *enable* change during *reset* is false? (Answer here: Then the unit behaves as described in 1 and 2).

References

[1] L. Claesen, editor. *Formal VLSI Specification and Synthesis VLSI Design Methods I and II.* IFIP, sponsored by IMEC, North–Holland, 1990.

[2] T. 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, 1995.

[3] L. Pierre. The formal proof of the 'min–max' sequential benchmark decribed in CASCADE using the Boyer–Moore theorem prover. In L. Claesen, editor, *Formal VLSI Specification and Synthesis VLSI Design Methods II*, pages 329–348. IFIP, sponsored by IMEC, North–Holland, 1990.