A DENOTATIONAL APPROACH TO LANGUAGE SPECIFICATION: A CAUSAL BLOCK DIAGRAM CASE STUDY

Ben Denckla Pieter J. Mosterman *
Hans Vangheluwe **

* *The MathWorks, Inc., Natick, MA 01760, USA*
** *McGill University, Montreal, Canada*

Abstract: In the design of embedded control systems, a variety of languages are used by different teams and in different development phases. Part of this variety comes from the use of domain-specific modeling languages that are tailored to the mental concepts of the user. This puts forward the need for efficient, systematic, and structured design of the modeling languages themselves. In particular, a precise and preferably executable specification of the language should be provided, although not all domain-specific language may be specified in this manner. A denotational semantics can provide such a specification. This paper presents a denotational semantics for a language BdAppLang. BdAppLang can be viewed as an abstract syntax for a simple subset of the domain-specific language of causal block diagrams, which are predominantly used in the system-level phase of control system design. The semantics are given in Haskell, and thus are executable while still conforming to the tradition of giving denotational semantics in a language derived from the *lambda calculus*.

Keywords: Domain-specific modeling, language design, block diagrams, denotational semantic

1. INTRODUCTION

Computational modeling is increasingly being used in industry to shorten the product development time. In one way, it is used to *complement* physical modeling, i.e., the design of physical prototypes and mock-ups. In an alternate application, it is used to *replace* physical modeling. Computational modeling has many advantages over physical modeling, as it is much easier and less costly to design a computational model. Furthermore, computational models are more flexible in their utility and modification.

For the design of embedded control systems, many different computational models, or 'models' for short, are being used. For example, models are used for designing the control laws, models are used for specifying the implementation of a controller, and models are used to improve the robustness of a controller (Mann, 1996; Mosterman *et al.*, 2004).

Because of the wealth of disciplines involved in the entire control system design process, the models that are used are designed by engineers with very different backgrounds and for many different purposes. For example, the control law design is often based on continuous time differential equation models because they are amenable to (automatic) synthesis methods. On the other hand, modeling the implementation of a controller is better done using discrete-time models, as they are closer to

the implementation of a controller on a microprocessor (Hyde, 2002).

Similarly, models based on finite element analysis, frequency domain methods, multi-body methods, state transition approaches, etc., may be employed in the design process. Each of these models relies on a different modeling language that is best suited for the problem at hand.

In general, a modeling language that is chosen to design a model should be closely aligned with the problem that needs to be addressed. So, it should have syntax that is close to the background and education of the users and the semantic notions that the language includes should be such that it allows elegant models (going by the premisse that elegant models are the best models). In case semantic notions require 'work-arounds' or some 'tweaking' or any usage that is poorly aligned with the mental concept of the model designer, the modeling process becomes error prone and a less useful if not incorrect model is bound to ensue.

To support this need for modeling languages that are best-suited for a particular problem, *domain-specific* languages are desired. Such languages can be tailored to the background of the model designers, the particular concepts that need to be captured, and the problem that needs to be solved by the model.

The design of modeling languages has been an active area of research, recently. In particular, the design of the syntax of a modeling language is well-understood. This is where the use of meta-modeling has proven very successful.

Meta-modeling concerns the modeling of models (Mosterman and Vangheluwe, 2000), or, in other words, the modeling of modeling languages. Once a model of a modeling language (the meta model) has been designed, editors and parsers can be automatically derived from the meta model. Examples of successful application of meta modeling are DoME (Engstrom and Krueger, 2000), the Generic Modeling Enironment (GME) (*Generic Modeling Environment*, 2002), and AToM3 [1].

Modeling the semantics of a language has proven to be much more difficult. Graph grammars have been successfully applied to model the semantics of graphical modeling formalisms either by (i) modeling a transformation from the newly defined language to an existing language with known semantics, or by (ii) modeling the execution of a model as a set of model transformations defined for the newly defined language (de Lara *et al.*, 2004). The first approach can be considered denotational in spirit, whereas the second is operational in nature.

[1] http://atom3.cs.mcgill.ca/

Though the graph grammar approach appears promising and holds great potential, it still falls short in terms of efficiency compared to dedicated implementations where the semantics are typically associated by hand crafting a compiler to transform the syntax into computable code.

The translation into an operational form is well-suited for model execution, but for understanding the intricacies of a language and its complexity, a denotational representation may be better. This is a more powerful way of reasoning about languages and comparing them.

Causal block diagrams are predominantly used in the design of control systems (e.g., (Åström and Wittenmark, 1984; Chen, 1970)) yet the semantics of industrial tools such as Simulink® (Simulink, 2004)) that support their use at times contain implicit assumptions that can complicate understanding by the user and language developer alike (Denckla and Mosterman, 2005). In this paper, a denotational specification for causal block diagrams (adopted from (Posse *et al.*, 2002)) is given, based on earlier work on language design (Schmidt, 1986) and a functional approach to languages design and analysis (Denckla and Mosterman, 2004; Nilsson *et al.*, 2003). The intent is to make the semantics explicit and unambiguous so as to arrive at a well-defined causal block diagram formalism.

In Section 2 causal block diagrams are introduced by their abstract syntax. In Section 3, an abstract syntax that is amenable to a denotational semantics specification is given. Section 4 presents a language to capture denotational semantics. Section 5 extends the language defined in Section 4 to support specification of block diagrams and presents Haskell as an approach to execution. In Section 6 the conclusions of this work are given.

2. CAUSAL BLOCK DIAGRAMS

Control system design is an involved process that moves through many different stages. For example, a model of the physical system to be controlled may be derived using finite element methods. This could result in a high order model, which needs to be reduced to a less complex version to allow synthesis and often numerical matrix methods are used for this (e.g., (Varga, 1999)) as available in, for example, MATLAB® (MATLAB, 2004).

Once the reduced order model is arrived at, control algorithms are developed to achieve the control objectives. The algorithms are then embedded into a model of the control system. This model embodies aspects such as mode switching between control laws for the different control regimes; data analysis computations such as scaling, calibration,

and filtering; and sample rate effects. Modeling the control *system* is often done using causal block diagrams with, for example, Simulink.

An example of a causal block diagram is given in Fig. 1. It shows two cascaded multiplier blocks, *mul1* and *mul2*. The first multiplier block, *mul1*, takes two values that are input to the block diagram, *u0* and *u1*, multiplies them, and the result of this is multiplied by a third input value, *u2*, by *mul2*. The result of the second multiplication is the block diagram output, *y*.

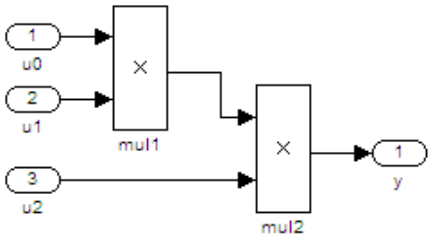

Fig. 1. A causal block diagram model.

This work will concentrate on the abstract syntax of block diagrams (the adjective causal will be omitted). The concrete syntax of block diagrams is irrelevant for the semantics analysis given in this paper.

A block diagram then consists of the following set of elements:

- *Blocks* are the basic elements of a block diagram.
- *Input ports* may be bound to blocks or be free. If they are free (e.g., `y` in Fig. 1), they are output of the block diagram, otherwise, they are input to the block they are bound to.
- *Output ports* may be bound to blocks as well. Similar to input ports, free output ports (e.g., `u0` in Fig. 1) are input to the block diagram while they are ouput to the block they are bound to otherwise.
- *Connections* are arrows from output ports to input ports.

In general, the behavior of a block may be defined in terms of another block diagram, thus creating hierarchy (Denckla and Mosterman, 2005). That notion, however, will not be elaborated as it is not important within the scope of this paper and would add significantly to the length of the treatise.

3. THE SYNTAX OF BLOCK DIAGRAMS IN BDAPPLANG

When a block diagram is interpreted, it may be translated in a number of abstract syntaxes, one following the other. The sequence of abstract syntaxes contain less and less information that is superfluous for the eventual interpretation.

In this work, the abstract syntax outlined in Section 2 is translated into another abstract syntax, which is then used for the denotational semantic specification. The abstract syntax consists of the following elements:

- A *block diagram*, BD, is a duple $BD = \langle blocks, outs \rangle$ that consists of a list of block characteristics, *blocks*, and a list of outputs, *outs*.
- A *block characteristic* is a triple $block = \langle name, func, inputs \rangle$ of block name, *name*, block function, $func$, and a list of block input arguments, *inputs*.

Blocks and diagrams may have multiple source ports, so block inputs and diagram outputs must specify port numbers. A special identifier, `toplevel`, exists for the top level input.

4. A LANGUAGE FOR DENOTATIONAL SEMANTICS

In order to rigorously describe behavior of causal block diagrams, a language is required to capture the denotational semantics. Here Haskell (Jones, 2003) is used in a style based on previous work on denotational semantics (Schmidt, 1986).

First the core language referred to as AppLang (as it is based on the Section *An Applicative Language* in (Schmidt, 1986)) is defined by giving the syntactic and semantic domains. In this work, a module `AppLang` ('denotational semantics block diagram core') is defined which exports the function `mPro` that assigns meaning to a program. It also exports the data types that `mPro` takes as input; expressions `Exp` and identifiers `Ide`. It further exports the data types that `mPro` produces as output; values `Val` and errors `Err`.

The *syntactic domain* then consists of programs, where a program `Pro` is an expression `Exp`. An expression can be one of six possible forms:

- `LamExp` is a lambda expression, a function definition (Slonneger *et al.*, 1995).
- `AppExp` is a function application.
- `LerExp` is a recursive *let* expression.
- `TupExp` is a tuple constructor.
- `IdeExp` is an identifier expression.
- `IntExp` is an integer constant.

The *semantic domain* consists of denotable values that can be one of four possible forms:

- A function with denotable value as domain and co-domain.
- A tuple of denotable values.
- An integer.

- An error that can be one of the following three forms:
  - an undefined identifier,
  - the application of a non-function, and
  - an incorrect argument type.

In summary, the syntactic and semantic domains are given by:

```
module AppLang
(mPro,Exp(..),Ide,Val(..),Err(..)) where

type Pro = Exp

data Exp =
    LamExp Ide Exp |
    AppExp Exp Exp |
    LerExp [(Ide,Exp)] Exp |
    TupExp [Exp] |
    IdeExp Ide |
    IntExp Integer

type Ide = String

data Val =
    FunVal (Val -> Val) |
    TupVal [Val] |
    IntVal Integer |
    ErrVal Err

data Err =
    UndefIdeErr Ide |
    AppNonFunErr Val |
    TypErr [Val] [Val]
```

The *semantic valuation functions* of AppLang consist of the following functions:

- `mPro` provides the meaning of a program. It takes an expression `p` and returns a value that is the meaning of the expression treated as a stand-alone program, i.e., the meaning in the initial environment `iniEnv`.
- `mExp` provides the meaning of an expression. It takes an expression `Exp` and an environment `Env` and returns a value `Val` that is the meaning of the expression in that environment. The function `mExp` takes the following forms:
  - `mExp (LamExp i x) e` returns the meaning of a lambda expression with argument `i` and body `x` in environment `e`.
  - `mExp (AppExp fx ax) e` returns the meaning of applying expression `fx` to expression `ax` in environment `e`.
  - `mExp (LerExp ds x) e = mExp x (fix newEnv)` returns the meaning of a *letrec* expression with declarations (identifier/expression pairs) `ds` and body `x` in environment `e`. The meaning of this *letrec* is the meaning of `x` in an environment that is the fixed point of the function `newEnv`.
  - `mExp (TupExp xs) e = TupVal $ mExps xs e` returns the meaning of a tuple expression with expression list `xs` in an environment `e` which is a tuple value of the meanings of the `xs` in `e`.
  - `mExp (IdeExp i) e = e i` returns the meaning of identifier `i` in environment `e`, which is the value that `i` is bound to in `e`.
  - `mExp (IntExp n) e = IntVal n` returns the meaning of an integer constant. The meaning of the expression is the corresponding integer value.

An additional evaluation function `mExps` operates on lists. Its mechanics are similar to those of `mExp`, but applied to a list of expressions.

The valuation functions can be summarized as:

```
mPro :: Pro -> Val
mPro p = mExp p iniEnv
mExp :: Exp -> Env -> Val
mExp (LamExp i x) e =
    FunVal (\v -> mExp x $ updEnvOne i v e)
mExp (AppExp fx ax) e =
    apply (mExp fx e) (mExp ax e)
        where
        apply (FunVal f) v = f v
        apply v _ = ErrVal $ AppNonFunErr v
mExp (LerExp ds x) e = mExp x (fix newEnv)
    where
    fix f = let x = f x in x
    newEnv e' = updEnvUnz dis (mExps dxs e') e
        where
        (dis,dxs) = unzip ds
mExp (TupExp xs) e = TupVal $ mExps xs e
mExp (IdeExp i) e = e i
mExp (IntExp n) e = IntVal n
mExps xs e = map (flip mExp e) xs
```

A number of 'helper' functions for the semantic (valuation) functions are created. These functions all operate on an environment:

- An environment `Env` is a function from an identifier `Ide` to a value `Val`.
- The function `updEnv` takes a list of bindings (identifier/value pairs) and an environment, and gives back this environment updated with the new bindings.
- The *maybe* call returns `(lookup i b)` if the lookup succeeds, otherwise it returns `e i`.
- A function `updEnvOne` updates an environment with a single binding.
- A function `updEnvUnz` updates an environment with 'unzipped' bindings of identifiers `is` to values `vs`.
- The initial environment `iniEnv` is the empty environment updated so that a few useful functions can be built into the language.
- The empty environment is one in which all identifiers are bound to an 'undefined identifier' error.

In summary:

```
type Env = Ide -> Val
updEnv :: [(Ide,Val)] -> Env -> Env
updEnv ivs e = \i -> maybe (e i) id (lookup i ivs)
```

```
4
5 updEnvOne :: Ide -> Val -> Env -> Env
6 updEnvOne i v e = updEnv [(i,v)] e
7
8 updEnvUnz :: [Ide] -> [Val] -> Env -> Env
9 updEnvUnz is vs e = updEnv (zip is vs) e
10
11 iniEnv = updEnv builtins empEnv
12
13 empEnv i = ErrVal $ UndefIdeErr i
```

The built-in operations "builtins", e.g., '+' and '*' are not of central significance to the semantics, and, therefore, they will not be listed here. The reader is referred to the appendix for details.

5. EXTENDING THE CORE LANGUAGE FOR BLOCK DIAGRAMS

Now that the denotational semantics of AppLang is available, it needs to be defined how a block diagram is mapped onto AppLang. To this end, an extension to AppLang is defined, called BdAppLang, that includes notions specifically included for defining block diagrams. It is shown how this mapping results in an executable representation of the block diagram.

5.1 Mapping a Block Diagram Onto BdAppLang

First, the extensions that constitute BdAppLang are defined as part of the `BdAppLang` module that imports all the `AppLang` definitions:

```
1 module BdAppLang
2 (
3 bdExp,Signal(..),
4 eq0,add,mul,neg,sbs,ifeq0,sbt,
5 lambda2,apply2,fstErr
6 ) where
7
8 import AppLang
9 import List
```

The function `bdExp` gives an expression for the block diagram described by block list `blocks` and output list `outputs`. It gives a lambda expression whose input identifier is `"toplevel"` and whose body is a *letrec* expression with the declarations `decls` and body `body` as argument. The `decls` are generated from the block list `blocks` using a function `getDeclForBlock`.

To obtain a list of declarations, the function `getDeclForBlock` takes as argument one block at a time, and it is iterated over the list of blocks by merit of the `map` function. For each block, `getDeclForBlock` then operates on the three constitutive parts of a block, i.e., its output `outid`, the function that describes the block behavior `func`, and the block inputs `inputs`. This corresponds to the abstract syntax defined in Section 3. The output is produced by using `AppExp` to apply the function to the inputs.

The `body` that is the argument to the *letrec* expression in `bdExp` is the output list `outputs` converted to a tuple expression by a function `signalsToTuple` which takes a list of signals `signals` and returns a tuple expression of subscripted identifiers.

A function `xSignal` is used to convert a bound signal described by an identifier `i` and a subscript `n`. It converts the signal to an application of the subscript operator `sbs` to the identifier. In case the identifier is not specified, the subscript operator is applied to the input of the block diagram, i.e., the unbound output ports in Section 4, referred to as `"toplevel"` in this paper. Moving from the block diagram abstract syntax to BdAppLang to AppLang, an output port (bound or unbound) becomes a `Signal` which becomes a subscripted identifier.

The basic set of functions that is needed to translate a block diagram into BdAppLang then becomes:

```
1 bdExp blocks outputs = LamExp "toplevel" $ LerExp decls body
2     where
3     decls = map getDeclForBlock blocks
4         where
5         getDeclForBlock (outid, func, inputs) =
6             (outid, AppExp func (signalsToTuple inputs))
7     body = signalsToTuple outputs
8     signalsToTuple signals = TupExp $ map xSignal signals
9         where
10        xSignal (Bound i n) = sbs (IdeExp i) (IntExp n)
11        xSignal (Unbound n) = sbs (IdeExp "toplevel") (IntExp n)
12
13 data Signal = Bound Ide Int | Unbound Int
```

5.2 Executing a Block Diagram

With BdAppLang defined, a block diagram can be conveniently expressed in a denotational specification. For example, consider the block diagram in Fig. 1. Its representation in BdAppLang is constructed using the following statements:

```
1 bd = bdExp blockList outputList
2     where
3     blockList = [
4         ("mul1", tupleMul, [Unbound 0, Unbound 1]),
5         ("mul2", tupleMul, [Bound "mul1" 0, Unbound 2])]
6     outputList = [Bound "mul1" 0, Bound "mul2" 0]
```

where

```
1 xfst id = ApplyExp (IdExp "fst") (IdExp id)
2 xsnd id = ApplyExp (IdExp "snd") (IdExp id)
3 tupleMul = LambdaExp "n" $ TupleExp [xmul (xfst "n") (xsnd "n")]
```

To execute this block diagram, the Haskell 98 based programming system Hugs 98[2] is used. Hugs 98 translates the denotational specification into an executable form. Executing this block diagram with input values [3, 4, 5] is done by the statement:

```
1 mPro $ AppExp bd (TupExp [IntExp 3, IntExp 4, IntExp 5])
```

[2] http://www.haskell.org/hugs/

which produces the expected output `Tup [Int 12,Int 60]`, i.e., a tuple of integers 12 and 60 (3*4 and (3*4)*5).

Note that this paper addresses the evaluation of a block diagram for one given set of input. It does not intend to provide an *execution engine* that will repeatedly evaluate a block diagram to generate a sequence of output values (which may be associated a temporal notion). In other work, the details of generating such an execution *trace* have been discussed (Denckla and Mosterman, 2005), which shows that the evaluation of the block diagram function and the generation of a trace can indeed be well separated. In particular, this can be achieved by making the state an 'implicit input' to the block diagram.

6. CONCLUSIONS

Support for domain-specific modeling is critical in the control system design process. Many different aspect of a system need to be modeled by engineers with widely differing backgrounds and for very different purposes. This requires tailored modeling languages that specifically address a particular domain, model designer background, and problem that needs to be solved by using the model.

To design such languages, it is critical to be explicit about the language semantics. A powerful way to capture the semantics of a language well is by translating it into a denotational representation. A denotational semantics avoids overspecifying a language as it specifies at a higher level of abstraction than, for example, an operational semantics.

This paper has concentrated on causal block diagrams, which is a domain-specific language widely used in addressing the system-level aspects of an embedded control system. A denotational semantics for such causal block diagrams was presented, thus providing a clear and unambiguous interpretation. Because based on Haskell, a freely available compiler allows execution of the denotational semantics.

One advantage of this approach is the explicit definition of the semantics, which makes underlying assumptions clear and facilitates further reasoning about the meaning of causal block diagrams.

An additional advantage is the restriction to a basic set of denotational concepts to be used for the definition of causal block diagram semantics. This restriction prevents the inadvertent implementation of semantics that are beyond what a declarative formalism such as causal block diagrams should embody, which could easily happen when specifying the semantics by imperative code. All this leads to an approach that is precise, not over-determined, and executable.

REFERENCES

Åström, Karl J. and Björn Wittenmark (1984). *Computer Controlled Systems: Theory and Design*. Prentice-Hall. Englewood Cliffs, New Jersey.

Chen, Chi-Tsong (1970). *Linear Systems Theory and Design*. Holt, Rinehart and Winston, Inc.. New York. ISBN 0-03-060289-0.

de Lara, Juan, Hans Vangheluwe and Pieter J. Mosterman (2004). Modelling and analysis of traffic networks based on graph transformation. In: *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004)* (Eckehard Schnieder and Géza Tarnai, Eds.). Braunschweig, Germany. pp. 120–127.

Denckla, Ben and Pieter J. Mosterman (2004). An intermediate representation and its application to the analysis of block diagram execution. In: *Proceedings of the 2004 Summer Computer Simulation Conference (SCSC'04)*. San Jose, CA.

Denckla, Ben and Pieter J. Mosterman (2005). Formalizing causal block diagrams for modeling a class of hybrid dynamic systems. In: *Proceedings of the IEEE Conference on Decision and Control*. Seville, Spain.

Engstrom, Eric and Jonathan Krueger (2000). A Meta-Modeler's Job is Never Done: Building and Evolving Domain-Specific Tools With DOME. In: *Proceedings of the IEEE International Symposium on Computer Aided Control System Design*. Anchorage, Alaska. pp. 83–88.

Generic Modeling Environment (2002). http://www.isis.vanderbilt.edu/Projects/gme.

Hyde, Rick A. (2002). Fostering innovation in design and reducing implementation costs by using graphical tools for functional specification. In: *Proceedings of the AIAA Modeling and Simulation Technologies Conference*. Monterey, CA.

Jones, Simon Peyton (2003). *Haskell 98 Language and Libraries*. Cambridge University Press. Cambridge, UK. ISBN-10: 0521826144.

Mann, Heřman (1996). A versatile modeling and simulation tool for mechatronics control system development. In: *1996 IEEE Symposium on Computer Aided Control System Design*. Dearborn. pp. 524–529.

MATLAB (2004). *The Language of Technical Computing*. The MathWorks, Inc.

Mosterman, Pieter J. and Hans Vangheluwe (2000). Computer automated multi-paradigm modeling in control system design. In: *Proceedings of the IEEE International Sympo-*

sium on Computer-Aided Control System Design. Anchorage, Alaska. pp. 65–70.

Mosterman, Pieter J., Janos Sztipanovits and Sebastian Engell (2004). Computer automated multi-paradigm modeling in control systems technology. *IEEE Transactions on Control System Technology*.

Nilsson, Henrik, John Peterson and Paul Hudak (2003). Functional hybrid modeling. In: *Lecture Notes in Computer Science*. Vol. 2562. Springer-Verlag. New Orleans, LA. pp. 376–390. Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages.

Posse, Ernesto, Juan de Lara and Hans Vangheluwe (2002). Processing causal block diagrams with graph-grammars in atom3. In: *Proceedings of the European Joint Conference on Theory and Practice of Software (ETAPS)*. Grenoble, France. pp. 23 – 34. Workshop on Applied Graph Transformation (AGT).

Schmidt, David A. (1986). *Denotational Semantics – A methodology for language development*. David Schmidt. 234 Nichols Hall, Kansas State University, Manhattan, KS.

Simulink (2004). *Using Simulink*. The MathWorks, Inc.. Natick, MA.

Slonneger, Ken, Kenneth Slonneger and Barry Kurtz (1995). *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc.. Boston, MA, USA.

Varga, Andras (1999). Selection of software for controller reduction. SLICOT Working Note.

7. APPENDIX

The following operations are defined:

```
builtins =
    [
    ("true",true),("false",false),("eq0",eq0),
    ("+",addi),("*",mult),("-",subt),
    ("subscript",subs)
    ]
    where
    true = FunVal (\v1->FunVal (\v2->v1))
    false = FunVal (\v1->FunVal (\v2->v2))
    eq0 = FunVal f
        where
        f (IntVal n) = if n==0 then true else false
        f v = ErrVal $ TypErr [v] [intExa]
    addi = arithOp (+)
    mult = arithOp (*)
    subt = arithOp (-)

    subs = binOp subscript
        where
        subscript (TupVal vs) (IntVal n) =
            vs !! (fromInteger n)
        subscript v1          v2         =
            ErrVal $ TypErr [v1, v2] [tupExa, intExa]

    tupExa = (TupVal [])
    intExa = (IntVal 0)

    arithOp = binOp . intOp
    intOp op = op'
            where
            op' (IntVal n1) (IntVal n2) = IntVal $ op n1 n2
            op' v1          v2          =
                ErrVal $ TypErr [v1, v2] [intExa, intExa]
    binOp op = FunVal f1
            where
            f1 v1 = FunVal f2
                where
                f2 v2 = op v1 v2
```