Thesis for the Degree of Doctor of Philosophy

Functional Programming Enabling Flexible Hardware Design at Low Levels of Abstraction

Emil Axelsson

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Göteborg
Sweden

Göteborg, 2008

Functional Programming Enabling Flexible
Hardware Design at Low Levels of Abstraction

Emil Axelsson
ISBN 978-91-7385-147-3

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 2828
ISSN 0346-718X

Technical Report no. 41D
Department of Computer Science and Engineering
Division of Software Engineering and Technology
Research Group: Functional Programming

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Göteborg, Sweden
Telephone: +46 (0)31 772 10 00

Printed at the Department of Computer Science and Engineering
Göteborg, Sweden, 2008

Abstract

Continuous down-scaling of sizes in VLSI circuits causes low-level electrical phenomena to become more and more prominent performance stoppers in modern chip technologies. This forces designers to work at a lower level of abstraction than desired in order to gain control over these effects. The most dominating contributors to low-level performance problems are the routing wires, which are used to connect the gates (computational units) on the chip. Hardware description languages (HDLs) – which naturally strive for generality and reusablility – tend to focus on higher levels of abstraction. This means that the low-level effects are not visible and thus very hard to control. To date, we are not aware of any HDL that allows control over the effects of routing wires in a really useful way, so when high-performance is crucial, designers are left to interfacing directly with CAD tools for physical chip design.

This thesis presents a flexible wire-aware design system called Wired and shows how it has developed from a preliminary idea to a relatively mature system that can now be tried on real-world challenges. The present Wired builds upon the existing HDL Lava, and extends it with: (a) finer control over geometry, (b) support for geometrical refinement, (c) more accurate performance models, (d) basic wire-awareness, and (e) support for descriptions with abstract signal flow.

Wired is implemented in the functional programming language Haskell. It has a simple modular implementation consisting of three separate main components: (1) The Lava library, for representing the structural circuit view, (2) a new monadic library for expressing layout, and (3) a new library for light-weight logical variables.

The system is used to improve the accuracy of an algorithm for searching for fast, low-power parallel prefix networks. More experiments are needed in order to reach industrially applicable results, but the main contribution here is the fact that physically aware design exploration is actually achievable at this level of flexibility. Although not yet tested on industrial problems, Wired seems like a very promising system that has a good potential of reducing the effort of generating high-quality layouts of complex circuits.

Publications

This thesis is partly based on the following publications:

A. Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired – a Language for Describing Non-Functional Properties of digital circuits. In *DCC '04: Int. Workshop on Designing Correct Circuits (satellite workshop of ETAPS)*. 2004.

B. Emil Axelsson and Mary Sheeran. Wired: Wire-Aware Circuit Description. In *TECHCON '05*. Semiconductor Research Corporation, 2005.

C. Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-Aware Circuit Design. In *CHARME '05: Correct Hardware Design and Verification Methods*. Springer, 2005.

D. Emil Axelsson, Koen Claessen, and Mary Sheeran. Using Lava and Wired for Design Exploration. In *DCC '06: Int. Workshop on Designing Correct Circuits (satellite workshop of ETAPS)*. 2006.

E. Matthew Naylor, Emil Axelsson, and Colin Runciman. A Functional-Logic Library for Wired. In *HFL '07: Int. Workhop on Hardware Design and Functional Languages (satellite workshop of ETAPS)*. 2007.

F. Matthew Naylor, Emil Axelsson, and Colin Runciman. A Functional-Logic Library for Wired. In *ACM SIGPLAN workshop on Haskell*. 2007.

G. Emil Axelsson. Flexible Hardware Design at Low Levels of Abstraction. In *DCC '08: Int. Workshop on Designing Correct Circuits (satellite workshop of ETAPS)*. 2008.

For publications C and F, the full paper has been reviewed. The remaining publications have been reviewed based on a shorter abstract. For publications D and G, there was no paper submission – only slides included in the participants' proceedings.

Chapter 2 is based on publication C, but also includes most of publication A, B and D. Chapter 3 is based on publication F, which is a revised and extended version of publication E. The oral presentations of publications E and F also included parts of chapter 4. Chapters 6 and 9 are based on publication G. The remaining chapters (7 and 8) contain work that is presented for the first time in this thesis.

Personal contributions

The following list describes the personal contributions to the different publications:

A,B,C,D:

The Wired system was conceived through discussions between Mary Sheeran, Koen Claessen and me. I made the implementation of Wired, including most design decisions. Mary developed the new network ("Slices"), and the description of Slices in Wired was done by Mary and me together. These publications are represented by chapter 2. In this chapter, section 2.3.1 was written by Mary, and the rest by me.

E,F: The logic programming library was developed jointly by Matthew Naylor and me. Most of the implementation was done by Matthew, who also wrote most of the text. I was mostly involved in writing sections 3.3.9, 3.4.3 and 3.6.

G: This is my work, except for the algorithm used in the case-study, which was developed by Mary Sheeran. The text was written by me.

Chapters 4, 7 and 8 are my own work.

Acknowledgments

First of all, I want to thank my supervisor, Mary Sheeran, for believing in me and always encouraging and helping me. In situations where I have started to doubt my ability to complete this work, Mary has always managed to turn the situation around and give me new confidence. I would also like to thank Koen Claessen for invaluable help and guidance – he has been like a second supervisor to me. Being supervised by two such great people has been a real privilege.

This work received Intel-custom funding from the Semiconductor Research Corporation (SRC). I am very grateful to Intel's Strategic CAD Lab (SCL) and SRC for this opportunity. I specially want to thank John O'Leary, Emily Shriver, Carl Seger and Steven Burns for professional guidance of this work. John and Emily also took good care of me on my two visits to SCL. I am also grateful for the support from Andy Martin at IBM and for the pleasant and productive time I had visiting IBM.

Thanks to Matthew Naylor who has had a great impact on this work. With his great sense for elegant programming, Matthew managed to bring order in my chaotic system. Also thanks to Colin Runciman and Matthew for letting me visit their department at the University of York.

Many thanks to Per Larsson-Edefors for valuable guidance. Per's expert knowledge in the VLSI field has helped me to stay focused on the actual problem. Thanks also to Magnus Själander for helping me with the local CAD tools.

Thanks to my thesis opponent Warren Hunt for valuable comments on an earlier draft of the thesis.

I want to thank my office mates, Hans Svensson and Joel Svensson, for great company at work. Hans and I have shared most of our time as Ph.D. students together, and I wish him good luck with his new job and his forthcoming marriage. Also thanks to my previous office mate Magnus Björk and to Andreas Larsson, and all the other colleagues at the department.

Finally, I want to thank those who make my life worth living: God, my family and my friends in Fiskebäck. I am especially grateful to my wife Arina who has supported me and taken care of our family while I have been working on this thesis.

Contents

Chapter 1

Introduction

This chapter aims to describe the problem background with focus on how effects of wires on the chip are obstructing advancements in modern VLSI designs. There is also a summary of related work. For explanation of some common terminology used throughout the thesis, see appendix A.

1.1 Abstraction in hardware design

Software programming can be done at different levels of abstraction. Languages like Haskell or Java offer a high-level programming environment, while assembler languages offer tedious low-level coding, possibly with more fine control over performance. Hardware description is also done at different abstraction levels, from high-level software-like environments to low-level composition of electronic building blocks. A major difference between hardware and software is that hardware design has a much wider range of abstraction levels. Typically, the higher levels are only concerned with functional aspects, while the lower levels take more and more physical aspects into account. Another important difference is that in hardware, all functional units operate completely concurrently.

At the border between high-level and low-level hardware design, we have the *netlist*, which describes a circuit as a network of connected logical gates (simple Boolean functions), but ignores any physical details of the circuit it represents. The netlist abstraction enables a powerful top-down design flow, where circuits are first described and verified functionally, and later transformed into a real hardware implementation of the same function. This transformation is done in several steps, most notably:

- **Technology mapping:** In an abstract netlist, the gates are just Boolean functions. The first step towards an implementation is to map each gate to a specific implementation in a *standard cell library*[1].
- **Placement:** This stage decides where each cell should be placed on the chip area.
- **Logistic routing:** This stage routes power supply and clock wires to the cells.
- **Signal routing:** This stage routes the signal wires, each one corresponding to a signal connection in the netlist.

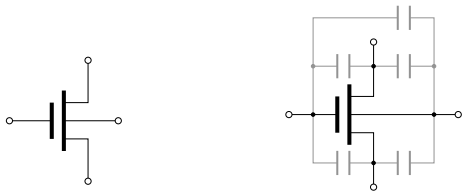
We refer to the netlist as being in the *structural domain* and the phases after placement and routing as being in the *physical* (or sometimes *geometrical*) *domain*. Higher-level descriptions, which are not concerned with the way the design is partitioned into its building blocks, are said to be in the *behavioral* or *functional domain*. Designers that do not deal with the physical domain are said to do *logic design* and designers working with physical aspects are said to do *physical design*.

1.1.1 Non-functional properties

In a pure top-down flow, these domains should be completely separated, allowing logic design without consideration of physical implementation details. In reality, however, this is not the case. The transformation into the physical domain introduces *non-functional properties* – behavioral or practical phenomena, that have nothing to do with the circuit's desired function. Among the non-functional properties, most attention is usually paid to *signal delay* and *power consumption*, which are the key parameters for measuring circuit performance. But also, for example, design area and manufacturability fall into the category of non-functional properties.

While these phenomena are not part of the desired circuit behavior, they are still highly affected by decisions made in the functional/structural domain. Design specifications are normally equally concerned with the functional and the non-functional aspect – it doesn't matter that a new microprocessor design is capable of doing extremely complex arithmetic if it can only run at 1MHz speed, for instance. This means that it is, in practice, impossible to separate the functional and physical concerns according to the original abstraction. Abstractions that fail to hide the underlying details are sometimes called *leaky abstractions*, and this is a common phenomenon that arises in many different areas [Spo02]. The only way to keep the top-down flow is to somehow bring the physical consequences up through the abstraction, for example, by means of *prediction* (section 1.2.1).

[1] A standard cell is an implemented gate. Normally, each Boolean function has several corresponding cell implementations for use in different situations.



(a) (b)

Figure 1.1: (a) Symbol for ideal NMOS transistor (b) High-frequency transistor model

When modeling electrical components, the difference between the "ideal" model and reality can often be explained by *parasitic loads*. Figure 1.1(a) shows the symbol for an ideal NMOS transistor. This symbol has a set of associated equations that describe the ideal relation between the voltages on the four terminals. However, when the transistor is used at high frequencies – for example, in a radio receiver or a high-speed microprocessor – the ideal model does not at all match reality. A better high-frequency model is obtained simply by attaching a number of parasitic capacitances on the terminals, as in figure 1.1(b) [Rab03]. Each capacitor can be viewed as a charge container. Changing a signal value corresponds to transporting charge in and out of the container, a process with naturally takes both time and energy. As a consequence, parasitic loads are a source of some of the non-functional properties that we consider.

1.1.2 Wires

The most evident difference between the netlist and the final hardware is the presence of *wires*. In the netlist, the wires are only implicitly present as logical connections between signals, but in the physical realization, the wires are real metal lines, which just like transistors, have associated non-functional properties. However, in traditional hardware modeling, wires were considered to be ideal connections [FM97, FHWS00, Ho03], an approximation justified by the fact that the non-functional properties of wires were usually negligible compared to the non-functional properties of gates. Under this model, circuit performance could be estimated quite accurately already at the netlist level – consequences of decisions at the netlist level were seen immediately without speculation about the physical domain.

This scenario has now changed drastically due to the continuous down-scaling of feature sizes in chip technologies. Active devices gain performance in this scaling process; smaller areas give lower capacitance, and shorter channels give lower driving resistance. Wires, on the other hand, do not follow the same development. Their non-functional properties are highly dependent on their

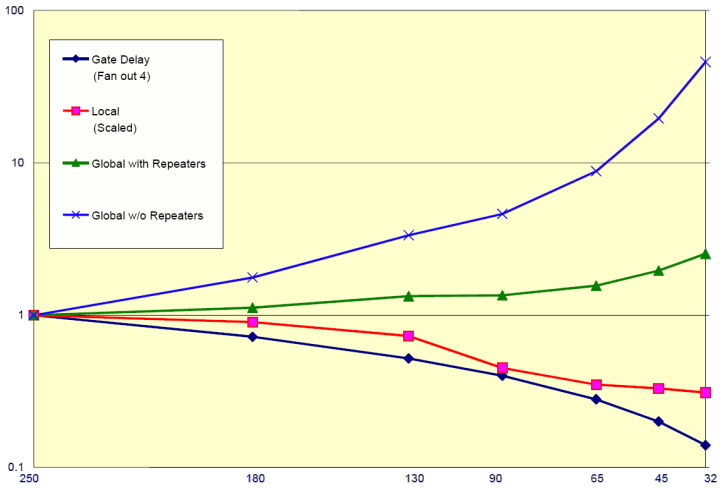


Figure 1.2: Relative delay for gates, local and global wiring versus feature size (source: [ITR05])

length, and since the chip size is usually constant, or even growing during scale-down, average wire length is not affected that much [Ho03]. Figure 1.2 shows the relative evolution of average delays for gates (lowest curve) and different classes of wires for different technology nodes. Only the local wires have a delay scaling that somewhat matches that of the gates. For a very careful treatment of on-chip wires and how they are affected by scaling, see Ho's dissertation [Ho03] and the related paper [HHM99]. A general overview of the different phenomena associated with scaling can be found in [SK98].

The most important thing to notice about scaling is the fact that wire properties are becoming more and more important compared to gate properties. For chip processes in the deep sub-micron (DSM) area ($< 0.25\mu m$), wire properties are, in fact, dominating. This is a major design problem, which has come to change the whole design flow. Plaza et al. put it this way:

> *"As interconnect plays a more significant factor in overall circuit delay, the focus of design methodology is shifting from logic optimization to interconnect optimization. While this transition has been occurring for over a decade, meeting performance objectives is becoming more and more difficult."* [PMB08]

More concretely, an EETimes article reports that around 75% of path delays, and up to 50% of the power consumption is due to wires in typical high-performance designs [Bro03].[2] This brings us to the conclusion that any method that aims

[2] According to [MKWS04], interconnects dissipate 50% of the *dynamic* power, which means that *total* interconnect power must be lower.

to combine the functional and physical domain needs to be highly aware of the interconnect wires.

1.2 Managing the wires

1.2.1 Wire load models

One common way to bring physical awareness up to the netlist level is to use wire load models (WLMs) [BKM01, SK98, GOPR02, SN00]. This allows individual wire load estimates to be obtained directly by looking at properties of the netlist. Normally, this estimation function is implemented as a lookup table mapping wire fanout and block size to estimated wire load capacitance. The lookup tables are obtained from wire statistics taken from similar previous designs.

A careful analysis in [GOPR02] reveals certain unavoidable inaccuracies in the wire load models (see also [Ho03]). The authors conclude that lookup models "are always inaccurate in a relative sense". Even estimations for very small blocks (around 20 cells) may result in large over- or underestimations of individual wire loads. On the other hand, Boese et al. argue that "even if WLMs are inaccurate, they might yet guide the tool to high-quality solutions" [BKM01]. In any case, this method is better than nothing, and it has been widely used, mainly in logic synthesis tools [SK98, GOPR02].

1.2.2 Constructive methods

Even with wire-load models, pure top-down design is not entirely feasible. What happens in practice is that the process has to be iterated several times between logic and physical design in order to converge towards the specification [BKM01]. This is usually called *design convergence* or *design closure*, and it is a major problem in practice. With each iteration, physical data is back-annotated so that prediction becomes more and more accurate, and the designers gain new insights into the circuit's nature; for example, which are the *real* critical paths, as opposed to the logical ones. This kind of iteration is normally very time consuming, especially since logical and physical design may be done by two completely separate working teams. In a competitive market, survival may depend on releasing new products before the competitors. Short product time-to-market may be a top-priority design goal, and avoiding costly design iterations is absolutely vital.

In order to overcome the inaccuracies of wire load models and thus speed up design convergence, there is a shift towards *constructive methods* [SN00], which instead rely on incomplete placement to estimate wire lengths.[3] In fact, in a

[3] Even constructive flows may use statistical wire load models in the initialization phase. [BKM01]

recent EETimes article, Erickson claims:

> *"Wireload models used to provide an easy way to supply an adequate amount of interconnect delay data to synthesis. Now that they have outlived their usefulness, ignoring interconnect is not the answer – we need to do a better job of modeling interconnect."* [Eri08]

He also proposes as solution to

> *"...provide the appropriate floorplan information to guide the interconnect model ... This requires more information than the traditional wireload model approach, but the benefits far outweigh the costs."*

Cong's work at UCLA is one example of a design flow in which interconnect is taken into account already from the start [Con01].

A simple constructive estimation method can use a partitioning-based placement strategy which stops when the blocks have reached a certain level of detail [RKG+92]. Other methods work in a bottom-up fashion, building small clusters of placed cells to use in the wire length estimations [ALES00]. Once approximate positions of blocks on the chip are known, wire lengths can be estimated by computing the minimal rectilinear spanning or Steiner tree [HRW92, GRSZ94] for all terminals connected by the wire, or if more accuracy is needed, by performing incomplete routing. Constructive methods are naturally more accurate than statistical ones, especially if the incomplete placement/routing mimics the behavior of the real tools that will be used. They also allow a trade-off between run time and accuracy by selecting the right level of detail for the incomplete placement and routing.

There is a new family of so-called *physical synthesis* tools [CKL+03, TKP+04, AKL+07], which are based around constructive estimation methods. The idea here is to integrate logic synthesis and physical design into one single tool in order to cut down on the costly design iterations. These tools typically perform simultaneous synthesis and placement on the same design. A common database containing information about all aspects of the design (logical, electrical and physical) is used for synchronization. Physical synthesis breaks the global design iterations into several smaller loops. If, for example, placement cannot proceed in some block of the design, it my be sufficient to re-synthesize only a small part of the netlist. Support for such incremental changes is an important part of reducing the design iterations [FHWS00].

IBM has a physical synthesis system called PDS, described in [TKP+04]. The authors claim that PDS, which has been used to generate more than 100 designs, "has reduced the time needed to achieve technology closure on a chip by 25%".

1.2.3 Layout-aware hardware description languages

When peak performance is needed, automatic CAD flows are often not good enough. Ho states that

> *"These highly automated CAD flows, used for designs such as graphics or network processors, generate structured netlists, place layouts from standard cell libraries, and route them together. The resulting designs can be completed very quickly – the Nvidia GeForce 4 graphics processor, with over 60 million transistors, went from initial concept to tapeout in just nine months – but do not clock as fast as the underlying technology permits..."* [Ho03]

Not surprisingly, the reason is claimed to be "discrepancies between pre-layout and post-layout wire loads". This situation means that microprocessor designers still do large parts of the designs manually [ADH+00, PAB+06]. On the other hand, the physical synthesis field is constantly progressing (see for example the recent award-winning paper [PMB08]), so the situation might well be changing in the future.

A particular type of circuits where manual design often has an advantage is regular or mostly regular structures, such as arithmetic modules.[4] In many such cases, there is a natural translation of the recursive structure into a corresponding layout which is more optimal than any automatic tool could have achieved [Sin00, ADH+00, CRX03]. Moreover, these circuits often perform very central operations and thus constitute a critical part of the overall circuit performance. So, spending extra time hand-crafting a central adder, say, may be a good investment that in the end allows a more rapid design convergence.

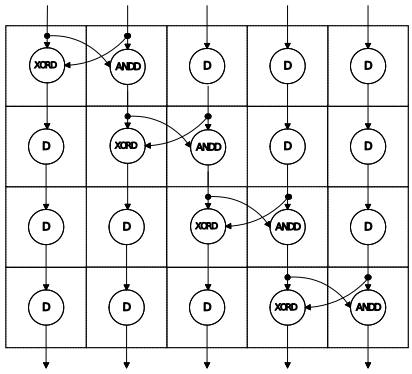
Most synthesis tools deal with arithmetic by having dedicated generators for common operations, such as addition and multiplication [Zim98]. They can produce modules for different data widths, and they also have the ability to choose between different architectures; for example, a large fast adder or a small slower one. This works well as long as we stick to designs that only use common operations; when extra functionality or special optimizations are needed, synthesis sooner or later reaches its limits. Furthermore, if absolute peak performance is required, it is not possible to design each module in isolation. Modules need to be fine-tuned to fit their electrical and physical context, and this usually requires manual intervention [ADH+00].

In order to control the exact layout of a design, one may use some kind of physical design tool. The interface to such tools is normally some kind of scripting language or a graphical editor. Designing through such low-level interfaces is tedious and makes sub-parts difficult to reuse. Therefore, it is often desired to have a higher-level interface to the tools, but without loosing control over the result. This is what *layout-aware design languages* offer.

[4] In fact, there are even indications that more regularity in general will be needed in the future in order to overcome challenges in manufacturability [McG06].

```
ABOVE FOR i = 0..(n-1) DO
 BESIDE (
  BESIDE FOR j = 0..(i-1) DO
   D [w(i,j)] [w(i+1,j)],
  XORD [w(i,i),w(i,i+1)] [w(i+1,i)],
  ANDD [w(i,i),w(i,i+1)] [w(i+1,i+1)],
  BESIDE FOR j = (i+2)..n DO
   D [w(i,j)] [w(i+1,j)] ) ;
```

(a)



(b)

Figure 1.3: Pipelined incrementer in Pebble (source: [LM98])

In layout-aware design languages, the user describes not only the circuit's function/structure, but also – in parallel – the layout of the functional units. For example, figure 1.3(a) shows the body of a pipelined incrementer definition in the layout-aware language Pebble [LM98]. If we remove the layout annotations (`BESIDE` and `ABOVE`), this is a standard VHDL-style structural description. With the annotations included, this code additionally specifies the placement shown in figure 1.3(b).

In general, a hierarchical netlist description can be made layout-aware simply by annotating each node in the hierarchy by the relative placement of its sub-components. As shown in figure 1.3(a), the annotations may specify horizontal or vertical placement. A description with such annotations completely determines the resulting floorplan, and this representation is called a "slicing floorplan" [LW01]. Singh's experiments on FPGAs show that design with this sort of user-provided relative placement constraints can give large improvements in performance [Sin00].

Since the designer has full control over structure and layout, the conceptual abstraction level in layout-aware languages is still quite low. However, this can now be mitigated using language-based abstraction techniques, such as recursion for capturing regularity, or higher-order functions for parameterization. The advantage of this kind of abstraction is that it is not leaky (section 1.1.1) in the sense that automatic synthesis is – the user still has a semantically clear connection to the physical nature of the circuit. The only missing information is the exact routing of the wires, but that can be estimated with resonable accuracy once the placement is known. Optionally, the language can control the wires as well, giving even more control and accuracy to the designer.

The increased low-level control means that specialized circuits, that are not tractable for synthesis with module generators, can now be done exactly the way we want them. However, for less regular circuits that are not so nicely recursive, the expressiveness is now much lower than in the behavioral domain, since we have to deal with much more detail here.

This thesis describes the evolution of a layout- and wire-aware design system, called *Wired*. The next section summarizes related work on layout-aware languages and functional hardware description languages in general.

1.3 Related work

There are several hardware description languages (HDLs) that allow structural hardware description with layout information encoded: ALI [LNS+82], μFP [She83], Zeus [GL85], CADIC [BHK+87, Dre92], Ruby [JS90, GL95, GL01], Lava [Cla00, CSS01], Hydra [O'D95, O'D04], Pebble [LM98, MLD02], Quartz [PL05] and IDV [Seg06], to mention a few. Most of these are not primarily intended as layout languages, but as languages for structural netlist description. However, as we have already pointed out, layout-awareness can be added quite easily to any structural HDL using relative placement annotations and slicing floorplan interpretation.

Recursion is a central mechanism in many of these languages (or alternatively iterative loops to achieve similar results). This makes it possible to capture regular circuits in concise descriptions, and to abstract away from parameters like data width. For example, here is a recursive description of a binary adder in Lava:

```
adder (cIn, [])        = ([],   cIn)
adder (cIn, (a,b):abs) = (s:ss, cOut)
  where
    (s,c)     = fullAdd (cIn, (a,b))
    (ss,cOut) = adder (c,abs)
```

The description works by recursion over the input lists. The two equations correspond to the base-case and the recursive case respectively. Each step applies a full-adder to the least significant bits, and recursively applies an `adder` to the remaining bits.

Another important mechanism, adopted from functional programming, is the use of higher-order functions to define *connection patterns*. Patterns provide a means of abstracting away from sub-components in the netlist and instead concentrate on reusable structural or geometrical patterns. As an example, Lava has a "serial composition" pattern `->-` capable of composing any two sub-circuits (with matching interfaces). For instance, an `and2`-gate can be defined as a composition of `nand2` and `inv`,

```
and2 = nand2 ->- inv
```

and we can also use that same pattern to compose two inverters:

```
buffer = inv ->- inv
```

Connection patterns can also be recursive. For example, `row` in Lava makes a horizontal row of the parameter component. The `row` pattern is reused in many common circuits, and can also be used in building larger patterns. For example, the `adder` above could be defined as

```
adder = row fullAdd
```

Apart from the additional layout constraints in `row`, the two definitions of `adder` are equivalent.

1.3.1 Embedded languages

Connection patterns are found in most of the languages mentioned above. Lava and Hydra are embedded in the functional programming language Haskell [J+03], which means that connection patterns come for free – they coincide exactly with higher-order functions, which form one of the corner stones of functional programming languages.

Being embedded in Haskell also gives the benefit of using Haskell as a meta-language. A Lava description is, in fact, a *circuit generator* – a Haskell program that, when run, produces a circuit netlist. The full power of Haskell can thus be used to make clever decisions about the circuit structure. A convincing example of this is Sheeran's work on so-called *clever circuits* which adapt themselves to functional or non-functional properties of their context [She03, She04]. Sheeran's work on circuits that are designed to be well-adapted to their contexts was in turn inspired by earlier work on delay- and layout-aware circuit synthesis [OVL96, SMOR98, UK02]. The work in this thesis aims to take more accurate account of wires than in these earlier approaches, while still having access to the power of Haskell in making decisions about circuit topology.

A final useful gain of the embedding in Haskell is static type checking, which is capable of finding a range of bugs (for example, some kinds of mismatching interfaces in compositions) at compile time. The usability of typed functional programming languages for embedded domain-specific languages has been explored and confirmed in [Hud98, Cla00].

One common type of interface mismatch that is not caught by Haskell's type checker is mismatching list lengths. This requires a stronger type system with support for sized types, such as in Cryptol [LM03], or a two-level approach, such as those mentioned in section 1.3.2.

An early example of an embedded layout language is found in "The SCHEME-79 Chip" [HJSB80]. The authors develop a complete chip capable of executing SCHEME code directly, and the circuit generator uses a layout language embedded in LISP.

Other examples of embedded hardware description languages are IDV [Seg06] and SystemML [Mar08]. IDV is a design and verification system developed at Intel's Strategic CAD Lab (who were funding this work). It contains an HDL embedded in their in-house functional language *reFL*^{ect}, and among other things, this language supports layout annotations. The ongoing development of IDV includes looking at ways to incorporate ideas from our work on Wired. SystemML is embedded in OCaml and targets high-level design with a clear link to the actual hardware.

1.3.2 Interaction with the host language

In Lava, the only thing we can do with a generator is to run it. Haskell is carefully desinged not to allow code inspection in order to maintain equational reasoning. This has implications on the kinds of circuit analyses that can be done. For example, the Lava system is linked with several BDD- and SAT-based verification engines that can be conveniently called directly from the Haskell code; however, this only allows verification of the generated netlists, not the generators themselves. So for example, a width-parameterized binary adder can only be verified for specific parameter values, not for all instances at once.

In contrast, the DE2 language, by Hunt et al. [HR05], is embedded in the ACL2 theorem proving system. This means that parameterized generators can be generally verified within the system. Their language also supports layout annotations, which suggests that it should be possible to do parameterized verification of layout generators. The language appears to be restricted to hierarchical statemachines, which is rather different from what our system targets. Still it remains an interesting research topic to see if ideas from DE2 can be used for validation of our kind of descriptions. DE2 originates from earlier work by Hunt on verified circuit generators as part of the FM9001 microprocessor verification effort [HB92].

Other systems exploit reflective capabilities of the host language to get access to the generators. Pace et al. use the aforementioned *reFL*^{ect} language for this [PT08]. Their intended applications include parametric verification and hardware compilers, and with the new support for layout annotations, this seems like a promising complement to our work.

Taha et al. use two-level language techniques (in the form of a Verilog preprocessor) to allow static checking of parameterized descriptions [AGM+07]. Future plans include having layout annotations in the types in order to get static guarantees on resource usage.

Along the same lines, Herrmann demonstrates how Template Haskell[5] can be used to obtain static guarantees about matching vector sizes for HDLs in Haskell

[5]Template Haskell is an extension to Haskell (implemented in the GHC compiler) for *compile-time* code reflection.

[Her06]. The same technique could potentially be used in Lava too, but there seems to be a certain syntactic and conceptual overhead imposed by Template Haskell, which suggests that this may only be worthwile for critical applications.

1.3.3 Relations

One problem with modeling circuits as functions (as Lava and Hydra do) is that connection patterns must be written for a particular signal flow direction. For example, Lava's `->-` has a distinct signal flow from left to right. It is possible to work around this by passing functions rather than signals through the pattern, but that leads to a very unnatural style of description. Instead, to allow different combinations of signal directions, O'Donnell finds it necessary to introduce different versions of each pattern in Hydra; array patterns like `fold` and `scan` come in unidirectional and bidirectional versions [O'D95]. Avoiding this kind of pattern proliferation was one of the reasons for making Ruby based on relations instead of functions.

Relations provide a more faithful modeling of real hardware and raise the design level by abstracting away from signal flow directions. The same pattern can be used as left-to-right, right-to-left or bidirectionally, and this was found to be a great simplification in the derivation of a multiplier circuit in Ruby [JS93]. The power of relational data flow abstraction has been further confirmed in work on the Quartz language [PL05]. The disadvantage of a relational model is that computations become more complicated. Simulating relational descriptions efficiently requires some kind of functional interpretation [Hut90].

1.3.4 Functional hardware description

Hardware design using functional programming languages (not necessarily layout-aware) has been an active research area for several decades. Some early influential work includes Sheeran's μFP [She83] and Johnson's Daisy [Joh84]. This thesis concentrates on design at low levels of abstraction, and on showing how functional programming can alleviate this kind of design. Most of the related work that we mentioned focuses on medium-level design (gate level or register transfer level). There are also a number of functional languages for *higher-level* design. Some examples are Bluespec System Verilog [Nik04], SystemML [Mar08] and SAFL [MS03].

1.4 Thesis overview

The system described in this thesis – Wired – should be seen as an extension to Lava which offers

- Finer control over geometry
- Geometrical refinement
- More accurate performance models
- Basic wire awareness
- Signal flow abstraction

These features are in no way unique for Wired, but their combination is! To our knowledge, no other system allows detailed placement and routing guidance in a setting where the meta-language can take advantage of layout-aware analysis to guide the circuit generation. Furthermore, Haskell's conciseness and a careful separation between structure and geometry make for very elegant and powerful descriptions. Lastly, Wired has a simple and modular implementation (the latest version, described in part II of the thesis) which can easily be extended with new functionality.

In earlier implementations (see chapter 2), there were several good reasons to use a completely relational circuit model:

- Circuit blocks had geometrical constraints which might be propagated in arbitrary directions (and not necessarily in the same direction as the signals).
- Wires were represented explicitly. Unlike gates, wires are naturally relational. In a functional setting, we would have to distinguish between a horizontal wire with left-to-right flow and one with right-to-left flow. A junction connecting four wires would have to come in four different versions, and so on. In a relational setting, each wire shape comes in one version only.
- Proper timing estimation considering wire effects requires bidirectional analysis methods (see section 2.2.2). In a relational setting, such analyses can be implemented simply using non-standard interpretation.

However, in the new implementation, these factors have disappeared, and so a functional model is more appropriate.[6] The only relational feature left is signal flow abstraction. In fact, this feature is even optional, and should only be used where needed.

1.4.1 Contributions

The thesis contributes to the areas *wire-aware circuit description* and *functional programming*. This section lists the contributions in detail.

[6] Actually, bidirectional analysis is still used (section 7.2), but this is now done separately and does not affect the circuit model exposed to the user.

Wire-aware circuit description

- We present Wired – a flexible and useful language for wire-aware circuit description. In Wired, a description starts off as purely structural and is then gradually refined by adding relative placement annotations to constrain the locations of gates. The main keys to the flexibility of the language are the separation between structure and geometry and Haskell's abstraction abilities, which allow the structure and geometry of common patterns to be captured in custom placement combinators.

  The geometrical view of the circuit, including wires, can be exported as a postscript picture, allowing a somewhat interactive development of the layout.

  Static timing analysis (a method for finding the highest speed at which a circuit can operate) gives quick feedback on circuit performance within the system. Furthermore, these estimations can be used to guide the construction of circuits, opening up for the development of optimizing circuit generators. Estimated delays are within 15% of those obtained from an industry-standard CAD tool.

  The wire awareness in Wired is two-fold:

  1. Timing analysis is based on accurately estimated wire lengths.
  2. Wire routing can be guided by specifying points that a given net should be routed through. We have not made any experiments where these have turned out to be helpful, but there is reason to believe that guides can be useful for longer regular wiring shapes (such as buses).

  The descriptions, including routing guides, can be exported and read into standard CAD tools for routing, analysis and fabrication.

- We have explored two methods for representing wires:

  1. **Explicitly**: In our first attempt at a wire-aware system, detailed routing was described explicitly by composing smaller metal segments into larger ones.
  2. **Implicitly**: In the system outlined above, wires are implicitly represented by the set of points that they connect.

  We found that the implicit model is vastly more flexible than the explicit model. In particular, the explicit model doesn't allow separation between structure and geometry, which has turned out to be crucial for a flexible system. So, comparing the two models, we have traded a minor decrease in accuracy for a major increase in flexibility.

- Signal flow abstraction allows common patterns to be described without regard to any particular signal flow. We have explored two approaches to acheive this in our system:

1. Using embedded logic programming
2. Using "circular programming" techniques

We found that, while both methods work, circular programming is simpler and gives a more clean and modular implementation.

- We have used the system to improve the accuracy of an algorithm for searching for optimal parallel prefix networks (commonly found in microprocessors). This was done by replacing an idealized cost function by one which uses Wired to perform static timing analysis on a completely placed circuit.

Functional programming

- We have developed a library for logic programming in Haskell. The library features an evaluation strategy called *residuation*, which allows efficient implementation of relations without backtracking, and this in turn enabled the first implementation of the aforementioned signal flow abstraction. In earlier experiments with an explicit wire representation (see above), we also used the library to express and solve geometrical constraints.
- We have used circular programming techniques to implement different circuit analyses (such as format conversion and timing estimation). This allows a more declarative programming style where only a single traversal of the structure is needed, even when the analysis has cyclic data dependencies. These analyses are built on the exact same library as is used for signal flow abstraction (see above).
- We have developed a Haskell library for expressing the layout of underlying computations. The library can be used on top of any kind of computation, and in the case of Wired, we use it on top of a purely structural circuit model. This modular design is what enabled the separation between structure and geometry for circuit descriptions.

1.4.2 Thesis structure

The rest of the thesis is divided into two parts. The first part presents the early work on Wired (called *Hardwired*), which uses a detailed circuit model with explicit wires. The second part presents a simplified, yet improved, system with an implicit wire representation. The first part is only included for historical reasons, and its main value is in the fact that it led to the system described in the second part.

Some readers may only be interested in the second part. Most places where part II depends on part I are clearly marked with references, so this way of reading the thesis should be possible with some amount of back-tracing.

Part I

Chapter 2 presents the initial idea and shows some examples. It also shows how a bidirectional RC-delay analysis can be expressed using non-standard interpretation.

In chapter 3, we develop a general library for logic programming in Haskell. This is motivated by the unsatisfactory earlier implementation of Hardwired's relational features. A minimal version of Hardwired is implemented in this library.

Chapter 4 presents the fully implemented Hardwired using the logic programming library. A multiplier reduction tree is described, and this example points at some fundamental weaknesses of the system.

The first part ends, in chapter 5, with a discussion about what was achieved, and what problems were encountered.

Part II

Chapter 6 starts with an introductory example to demonstrate the new circuit model developed in the second part of the thesis. After that, it gives an introduction to some standard monadic Haskell libraries that will be used in the succeeding chapters.

Chapter 7 presents a simpler replacement of the logic programming library from chapter 3 as well as a simple Lava library. This enables simple implementations of various, possibly bidirectional, circuit analyses, and two such examples are demonstrated.

Chapter 8 presents a general monadic library for expressing the layout of computations. It ends with an example of how to express the layout of arithmetic computations.

Chapter 9 shows how a much improved version of Wired can be implemented by combining the libraries from chapters 7 and 8. It also demonstrates how circuit descriptions can be analyzed and exported to CAD tools. The final case-study uses Wired to improve the accuracy of an algorithm for finding fast, low-power parallel prefix networks.

Part III

The thesis ends with a summary and some ideas for future work.

1.4.3 Prerequisites

This thesis spans over a wide range of topics: from electronics and digital circuits to functional programming and language design. Most of the thesis should be possible to read with a rather basic knowledge in the areas mentioned. However, some of the more detailed sections may require deeper knowledge. For example, parts of section 2.2 assumes an electronic background beyond the basics, and sections about implementation details require good knowledge of Haskell. The intention is that most example code should be possible to follow even with a mere intuitive understanding of Haskell. A quick introduction to Haskell is given in [HF92]. Monads, which are used a lot in the thesis, are explained in sections 3.2 and 6.3. Information about standard Haskell libraries and functions can be found through the brilliant search engine Hoogλe [Hoo]. As a general background to our style of circuit description, see [Cla00].

Most programming code given in the thesis is actual Haskell code that has been typeset using the `lhs2TeX` system [LHS]. The document source is a set of literal Haskell files, so most of the document code has even been compiled and tested. Compilation was done using the Glasgow Haskell Compiler (GHC) version 6.8.2. Chapter 3 requires the compiler options

```
-fglasgow-exts
-fno-mono-pat-binds
```

The chapters after that require

```
-fglasgow-exts
-fallow-undecidable-instances
-fallow-overlapping-instances
-fno-monomorphism-restriction
```

Part I

Hardwired

Chapter 2

Hardwired

This chapter presents our early work on Wired. Apart from sections 2.4 and 2.5, the chapter is a slightly revised and cut-down version of our paper at CHARME '05 [ACS05]. This chapter is mainly interesting for historical reasons. The current implementation of Wired (chapter 9) is very different from the one presented here, but this early work was an important step towards a much improved system. Because of the big difference, both in the front-end and the implementation, this earlier system has later been renamed to "Hardwired". The chapter ends with a retrospective discussion.

2.1 The Wired system

2.1.1 The core language

Wired is built around *combinators* with both functional and geometrical interpretations. Every description has a *surface* and a *surface relation*. A surface is a structure of contact segments, where each contact may or may not carry a signal. This structure specifies the interface of the description and keeps track of different signal properties. When flattened, it can represent the description's geometrical appearance. Figure 2.1(a) shows an example of a simple two-input and-gate. This is a 2-dimensional circuit, so the surface consists of four *ports*. The left- and right-hand ports are i-contacts ("insulators" – contacts without signals) of size 2. The inputs, on the top, are two size 1 s-contacts (contacts with signals). The output, on the bottom, is a size 2 output signal. This gate has a clear signal flow from input to output, which is not always the case for Wired circuits.

Figure 2.1: (a) Two-input and-gate (b) Beside composition, `d1*||*d2`

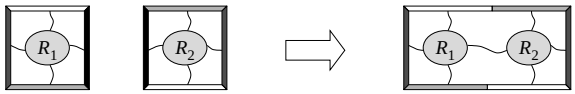


Figure 2.2: Combination of sub-descriptions

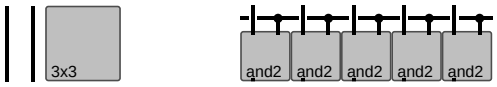
The surface relation relates parts of the surface to each other. It can capture both structural and functional constraints. A structural relation can, for example, specify that the number of inputs is equal to the number of outputs, and a functional relation could specify that two signals are electrically connected.

Wired is embedded in the functional programming language Haskell. The data type that is used to represent descriptions internally is defined as:

```
data Description = Primitive Surface Relation
                 | Combined  Combinator Description Description
                 | Generic   Surface (Surface -> Maybe Description)
```

A description is either a *primitive* defined by a surface and a relation, or a *combination* of two sub-descriptions. We will look more at *generic* descriptions in section 2.1.2. The `Combinator` describes how the two sub-surfaces are combined into one and indicates which surface parts are connected where the two blocks meet. This implicitly defines a new surface and relation for the combined description. Figure 2.2 illustrates a combination of two (not necessarily primitive) 2-dimensional circuits with relations R_1 and R_2.

The combinator `*||*` ("beside") places the first block to the left of the second, while `*=*` ("below") places the first block below the second. Figure 2.1(b) illustrates `d1*||*d2`. Note how the resulting top and bottom ports are constructed. The top ports of the sub-circuits are named `pB1` and `pB2`, and the resulting top port becomes the pair `com [pB1,pB2]`. The same holds for the side ports when using `*=*`. We will also use variations of these which have the same geometrical meaning but combine the surfaces differently. The variant `*||~` does the "cons" operation; if `d1` has port `pB1` and `d2` has port `com [pB21,pB22, ...]`, then the resulting port becomes `com [pB1,pB21,pB22, ...]`. The variant `~||*` does "cons" at the end of the list, and `~||~` and `-||-` are two variations of the "append" operation.



(a) (b)

Figure 2.3: (a) Layout after instantiation of `example1` (b) Layout of 5-bit bit-multiplier

The surface structure may be partially unknown. For example, a wire (wires are normal descriptions, just like anything else) need not have a fixed length, but can be instantiated to whatever length it needs to have. Such instantiation is done – automatically by the system – by looking at the surrounding context of each sub-description. The surrounding surfaces form a so-called *context surface*, and we require, for all sub-descriptions, that the surface and the context surface are structurally equal. This means that if, for example, a stretchy wire is placed next to a block with known geometry, the wire will automatically be instantiated to the appropriate length. The wire also has a relation that states that its two sides have the same length. So, if we place yet another wire next to the first one, size information will be propagated from the block, through the first one and over to the new wire. In Wired, this circuit is written:

```
example1 = wireY *||* wireY *||* block3x3
```

`wireY` is a thin vertical wire with unknown length, and `block3x3` is a pre-defined block of size 3×3 units. Instantiating this description and asking for a picture (through an interactive system) gives the layout in figure 2.3(a).

Note that `example1` relies on abstract data flow, as discussed in sections 1.3.3 and 1.4. Size information happens to be flowing from right to left, and the signal flow through the wires is not yet known – it depends on the context in wich the circuit is placed.

In Lava, circuits are constructed by just running their Haskell descriptions, so most of the instantiation procedure comes for free, from Haskell. Since Wired is relational, we cannot use the same trick here. Instead we have a separate *instantiation engine*, which is implemented in Haskell. This engine works by fix-point iteration – it traverses the description iteratively, propagating surface information from contexts to sub-descriptions and instantiating unknown primitive surfaces, until no more information can be gained.

2.1.2 Generic descriptions and connection patterns

In `example1` we saw wires that adapted to the size of their context. This is very useful since the designer doesn't have to give all wire sizes explicitly when writing the code. Sometimes we want to have sub-blocks whose entire content adapts

to the context. For this we use the final constructor in the definition of descriptions (section 2.1.1): `Generic`. A generic description is defined by a surface and an *instantiation function*. As the type (`Surface -> Maybe Description`) indicates, this function reads its current surface and may choose, depending on that information, to instantiate to a completely new description. Since this is a normal function on the Haskell level, it is possible to make clever choices here. For example, in the context of non-functional analysis (section 2.2), we can choose between two different implementations depending on some estimated value.

The most common use for generic descriptions is in defining *connection patterns*, which capture commonly used regular structures. The simplest connection pattern is the *row*, which places a number of copies of the same component next to each other. We can use it to define a bit multiplier circuit, for example:

```
bitMult = row and_bitM
  where and_bitM = and2 *=* (cro *||* crT0)
```

The primitives used are: `and2`, an and-gate with the surface from figure 2.1(a), `cro`, two wires crossing without connection and `crT0`, a T-shaped wire connection. Figure 2.3(b) shows the layout of this circuit instantiated for 5 bits.

We define `row` as follows:

```
row d = generic "row" xpSurf (row_inst d)

row_inst d surf = do len <- lengthX surf
                     case len of N 0 -> newInst thinEmptyY
                                 N _ -> newInst (d *||~ row d)
                                 _   -> noInst
```

The pattern is parameterized by a description `d`, and has unknown initial surface (`xpSurf`) and instantiation function `row_inst` (also parameterized by `d`). The instantiation function looks at the current surface and does a case analysis on its horizontal length. If the length is known to be 0 (the constructor `N` means known), the row becomes a thin empty block. This is the base-case in the recursive definition of `row`. For lengths greater than 0, we place one copy of `d` beside another row, using the `*||~` combinator. If the length of the context has not yet been resolved (the last case), we do not instantiate.

A simpler alternative to the above definition of `row` is

```
rowN 0 _ = thinEmptyY
rowN n d = d *||~ rowN (n-1) d
```

This definition takes an extra length parameter `n`, and does the whole unrolling on the Haskell level instead, before instantiation. This is both simpler and runs faster, but has the down-side that the length has to be known in advance. In the normal `row`, instantiation automatically resolves this length.

Generic descriptions or partially unknown surfaces are only present during cir-

cuit instantiation. After instantiation, when we want to view the layout or extract a netlist, we require all surfaces to be complete, and that all generic parts have been instantiated away.

2.1.3 Signal interpretation

Surfaces are structures of contact segments. A contact is a piece of area that may or may not carry a signal. However, it is possible to have more information here. The signal can, for example, be decorated with information about whether it is an input or an output, and about its estimated delay. This allows an analysis that checks that two outputs are never connected, and a way to compute the circuit's delay from input to output. We want to separate the description from the interpretation that we will later give to it, so that the same description can be used with different interpretations. This is done by abstracting the signal information on the Haskell type level. That is, we parameterize the `Description` type by the signal type `s`. This type variable can be kept abstract as long as we want, but before we instantiate the description, `s` must be given a particular type. At the moment, possible signal types are:

`NoInfo`	No information, just a structural placeholder
`Direction`	Signal direction (in/out)
`UnitTime`	Delay estimation under unit-delay model
`Resistance`	Output driving resistance
`Capacitance`	Input load capacitance
`Time`	Accurate RC-delay estimation

The operator `:+:` combines signal types. Such combinations are needed since it makes no sense to talk about delays if there is no notion of direction, for example. To increase readability, we define some useful type macros. For example,

```
type Desc_RCDelay =
       Description (Direction:+:Resistance:+:Capacitance:+:Time)
```

2.2 Non-functional analysis

2.2.1 Direction and unit-delay

Wired is a relational language, and is thereby not bound to any specific direction of signal flow. Still, most circuits that we describe are functional, so we need to be able to check that a description has a consistent flow from input to output. Here we use the signal interpretation with direction. While it is usually known for gates which signals are inputs and outputs, wire junctions normally have undefined directions initially. However, we know that if one signal in a junction is an input (seen from inside the junction), all the others must be outputs, in order to avoid the risk of short circuit. This constraint propagation can be

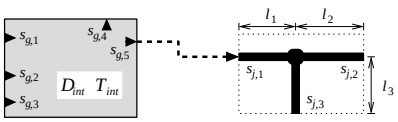


Figure 2.4: Gate and wire junction

baked into the circuit relation, and this is enough to help the instantiation engine resolve all directions (or report error). Figure 2.4 shows an example of a gate cell and a wire junction. Signal $s_{j,1}$ of the junction is indirectly connected to gate output $s_{g,5}$. If we assume that directions are propagated correctly through the intermediate wires, the context will eventually constrain $s_{j,1}$ to be an input, and by direction propagation, $s_{j,2}$ and $s_{j,3}$ will be constrained to outputs.

This is a simple example of circuit analysis by non-standard interpretation (NSI) [Luk90, Sin92, JN94, O'D95]. That is, instead of the standard interpretation of signals as Boolean values, we evaluate the circuit using some other signal type that is able to record some aspect of the circuit's nature. The important thing is that we can use normal evaluation for this – no external program is needed for analysis by NSI.

The simplest model for circuit delay estimation is the *unit-delay* model, in which each stage just adds a constant unit to the input delay – independent of electrical properties, such as signal drive strength and load. This gives a rather rough estimate of circuit delay.

As with directions, unit-delay can be resolved by the instantiation engine, provided that delays are propagated correctly through gates and wires. The gate in the above example has intrinsic unit-delay D_{int} (and an accurate time delay T_{int}, which will be used in the next section). D_k refers to the unit-delay of signal s_k. As instantiation proceeds, delay estimates of the input signals will become available. Delay propagation can then set the constraints

$$D_{g,4} = D_{g,5} = max[D_{g,1}, D_{g,2}, D_{g,3}] + D_{int}$$

The model can easily be extended so that different input-output paths have different delays.

For the wire junction, we want to capture the fact that longer wires have more delay. This dependency is hidden in the function *conv*, which converts distance to unit-delay. By choosing different definitions for *conv*, we can adjust the importance of wire delays compared to gate delays. Once the delay of $s_{j,1}$ becomes available, the following propagation can be performed:

$$D_{j,k} = D_{j,1} + conv(l_1 + l_k) \quad \text{for } k \in [2, 3]$$

These two propagation methods work for all wires and gates, independent of number of signals and logical function, and they are part of the relations of all

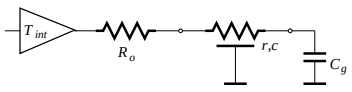


Figure 2.5: Circuit stage from output to input

wire and gate primitives. Since information is only propagated from inputs to outputs, this is a *forwards analysis*. In the next section, we will use a combination of forwards and backwards analysis.

2.2.2 RC-delay

For a more accurate timing analysis, we use the model in figure 2.5 [HE00]. A gate output is a voltage source with intrinsic delay T_{int} and output resistance R_o. A wire is a distributed RC-stage with r and c as resistance and capacitance per length unit respectively. Gate input is a single capacitance C_g.

A signal change on an output gives rise to a signal *slope* on connected inputs. This slope is characterized by a *time constant*, τ. For output stages with equal rise and fall times (which is normally the case), it is standard to define wire delay as the time from an output change until the input slope reaches 50% of its final value. For a simple RC-stage, see figure 2.6(a), the time constant is given by $\tau = RC$. The delay from the left terminal to the capacitor is then approximately equal to 0.69τ. For a distributed RC-stage (figure 2.6(b)) with total resistance $R = rL$ and capacitance $C = cL$, it can be shown that $\tau_{dist} \approx RC/2$ [Rab03].

Figure 2.6(c) shows a fanout composition of n RC-stages. The so-called Elmore delay [Rab03, GTP97, RPH83] from the left terminal to capacitor C_i can be approximated by a simple RC-stage with the time constant

$$\begin{aligned}\tau_{1,i} &= R_1 \cdot \left[\sum_{l\in[1..n]\setminus i} C_l\right] + (R_1 + R_i)\cdot C_i \\ &= R_1C_1 + R_1 \cdot \left[\sum_{l\in[2..n]} C_l\right] + R_iC_i \\ &= \tau_1 + R_1 \cdot \left[\sum_{l\in[2..n]} C_l\right] + \tau_i \end{aligned} \tag{2.1}$$

This formula also holds for distributed stages – R_i and C_i are then the total resistance and capacitance of stage i – or for combinations of simple and distributed stages. Note that the local time constants τ_1 and τ_i are computed separately and added, much as unit-delays of different stages were added. What is different here is the extra fanout term, where R_1 is multiplied by the whole load

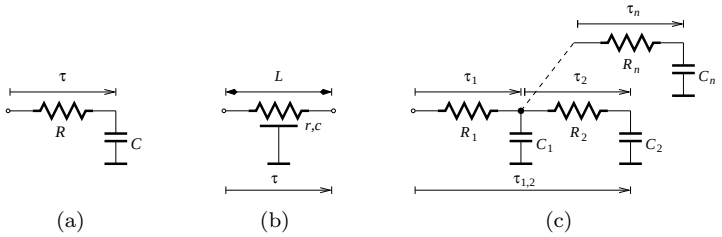


Figure 2.6: (a) RC-stage (b) Distributed RC-stage (c) Composition of RC-stages

capacitance. It is generally the case that the stages on the right-hand side are themselves compound; the RC-stage is merely an approximation of their timing behavior. Therefore, load capacitance needs to be propagated backwards from the load, through the stages and to the junction we are currently considering. So, for RC-delay analysis, we need a combination of forwards and backwards analysis. This is, however, a simple matter in a relational system like Wired.

We describe gate and wire propagation from the example in figure 2.4. Gates always have known output resistances and input capacitances, so no such propagation is needed. Therefore RC-delay propagation through gates behaves just like for unit-delay. Propagation through wire junctions is more tricky. We use R_k, C_k and τ_k to refer to the resistance, capacitance and RC-delay of the signal s_k. We also define R'_k, C'_k and τ'_k as the local resistance, capacitance and time constant of the piece of wire going from s_k to the junction. R'_k and C'_k can be computed directly from the corresponding length l_k, and $\tau'_k = R'_k C'_k/2$. The drive resistance and time constant of $s_{j,1}$, and the load capacitance of $s_{j,2}$ and $s_{j,3}$ will be resolved from the context.

The total load capacitance at the junction is given by

$$C_{junc} = C_{j,2} + C_{j,3} + C'_{j,2} + C'_{j,3}$$

Backwards capacitance propagation can then be done directly by the constraint

$$C_{j,1} = C_{junc} + C'_1$$

From equation 2.1 we get the time constant at the junction as

$$\tau_{junc} = \tau_{j,1} + \frac{R_{j,1}C'_{j,1}}{2} + \tau'_{j,1} + (R_{j,1} + R'_{j,1}) \cdot C_{junc}$$

Finally, forwards resistance and RC-delay propagation is done as

$$R_{j,k} = R'_{j,k} \quad \text{and} \quad \tau_{j,k} = \tau_{junc} + \tau'_{j,k} \quad \text{for } k \in [2,3]$$

Now, to perform an RC-analysis of a circuit, say `bitMult` from section 2.1.2, we first select the appropriate signal interpretation:

```
bitMultRC = bitMult :: Desc_RCDelay
```

This description is then instantiated in a context surface that specifies resistance, capacitance and delay on the inputs or outputs of the circuit.

2.3 Parallel prefix circuits

A modern microprocessor contains many *parallel prefix circuits*. The best known use of parallel prefix circuits is in the computation of the carries in fast binary addition circuits; another common application is in priority encoders. There are a variety of well-known parallel prefix networks, including Sklansky [Skl60] and Brent-Kung [BK82]. There are also many papers in the field of circuit design that try to systematically figure out which topology is best for practical circuits. We have been inspired by previous work on investigating the effect of wire delay on the performance of parallel prefix circuits [HE00]. Our aim has been to perform a similar analysis, not by writing a specialized simulator, but by describing the circuits in Wired and using the instantiation engine to run RC-delay estimations.

2.3.1 The parallel prefix problem

Given n inputs, $x_1, x_2, \ldots, x_n$, the problem is to design a circuit that takes these inputs and produces the n outputs

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_1 \circ x_2 \\ y_3 &= x_1 \circ x_2 \circ x_3 \\ &\vdots \\ y_n &= x_1 \circ \cdots \circ x_n \end{aligned}$$

where $\circ$ is an arbitrary associative (but not necessarily commutative) binary operator. One possible solution is the *serial prefix circuit* shown schematically in figure 2.7(a). Input nodes are on the top of the circuit, with the least significant input (x_1) being on the left. Data flows from top to bottom, and we also count the stages or levels of the circuit in this direction, starting with level zero on the top. An operation node, represented by a small circle, performs the $\circ$ operations on its two inputs. One of the inputs comes along the diagonal line above and to the left of the node, and the other along the vertical line from the top. A node always produces an output to the bottom along the vertical line. It may also produce an output along a diagonal line below and to the right of the node. Here, at level zero, there is a diagonal line leaving a vertical wire in the absence of a node. This is a fork. The *serial prefix* circuit shown contains 7 nodes, and so is said to be of *size* 7. Its lowest level in the picture is level 7, so the circuit

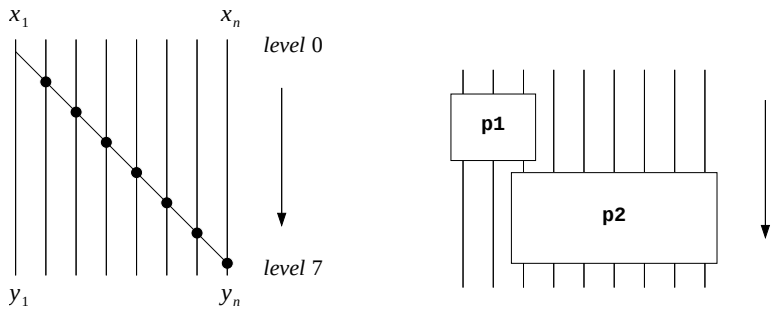


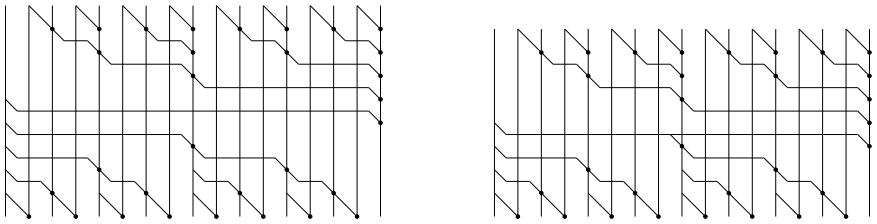
Figure 2.7: (a) Serial prefix (b) Prefix composition

has *depth* 7. The fanout of a node is its out-degree. In this example, all but the rightmost node have fanout 2, so the whole circuit is said to have fanout 2.

Examining figure 2.7(a), we see that at each non-zero level only one of the vertical lines contains a node. We aim to design *parallel* prefix circuits, in which there can be more than one node per level. For two inputs, there is only one reasonable way to construct a prefix circuit, using one copy of the operator. Parallel prefix circuits can also be formed by composing two smaller such circuits, as shown in figure 2.7(b). Repeated application of this pattern (and the base case) produces the serial prefix circuit.

For 2^n+1 inputs, one can use so-called forwards and backwards trees, as shown in figure 2.8(a). We call a parallel prefix circuit of this form a *slice*. At the expense of a little extra fanout at a single level in the middle of the circuit, one can slide the (lower) backwards tree up one level, as shown in figure 2.8(b). Composing increasing sized slices gives the well-known Brent-Kung construction [BK82] shown in figure 2.9. A shallower n-input parallel prefix circuit can be obtained by using the recursive Sklansky construction, which combines the results of two separate $n/2$-input parallel prefix circuits [Skl60], as shown in figure 2.10.

We have studied new ways to combine slices like those used to build Brent-Kung. By allowing the constrained use of fanout greater than 2, we have found a way to make slices in which the forward and backwards trees have different depths, and this leads to parallel prefix structures that have a good balance between small depth, few operators and low fanout. An example of such a structure is shown in figure 2.11.



(a) (b)

Figure 2.8: (a) Parallel prefix construction using a forwards and a backwards tree (b) The same construction with the lower tree slid back by one level

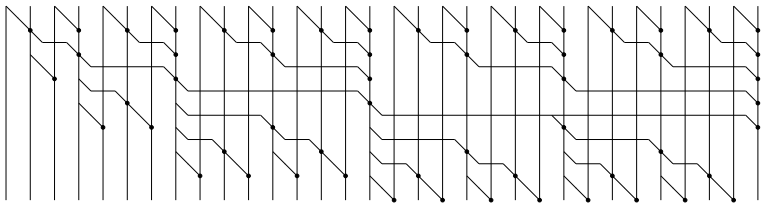


Figure 2.9: Brent Kung for 32 inputs

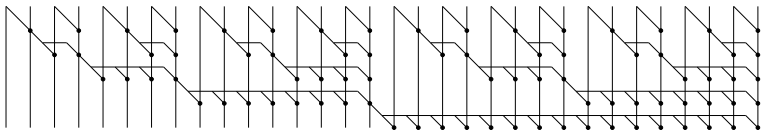


Figure 2.10: Sklansky for 32 inputs, with fanout 17

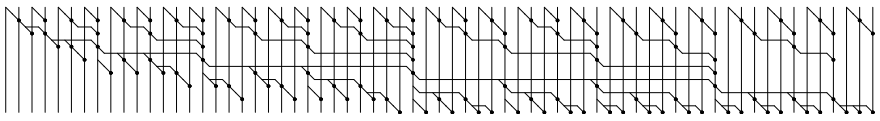
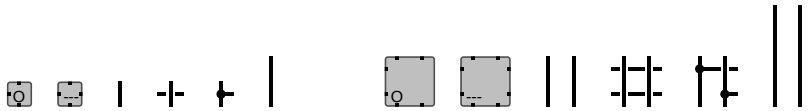


Figure 2.11: A new arrangement of composed slices, with 67 inputs, depth 8 and fanout 4



(a) (b)

Figure 2.12: Parameters `d`, `d2`, `w1`, `w2`, `w3` and `w4` for (a) 1-bit and (b) 2-bit computations

2.3.2 Wired descriptions

All of our parallel prefix circuits can be built from the primitives in figure 2.12(a). `d` is the operator with inputs on the left and top ports, and output on the bottom. Its companion `d2` additionally feeds the left input over to the output on the right-hand side. Although `d2` is a primitive, it behaves as if there were a horizontal wire crossing on top of it. `w1`, `w2` and `w3` are unit-size wire cells, and `w4` is a wire with adaptive length. Instead of hard coding these into the descriptions, we will supply them as parameters, which allows us to use the same pattern with a different set of parameter blocks.

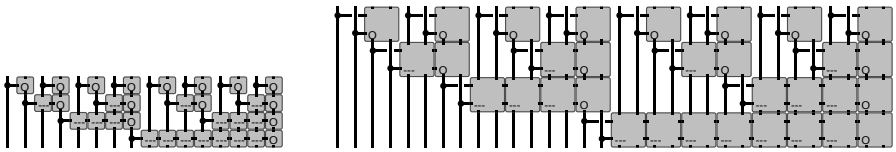
Just like for the row in section 2.1.2, we can choose between unrolling the structure during instantiation, or in advance on the Haskell level. Here we choose the latter, since it gives us more readable descriptions.

As shown in figure figure 2.10, Sklansky consists of two smaller recursive calls, and something on the bottom to join their results. This leads to the following description in Wired:

```
sklansky 0   = thinEmptyX1
sklansky dep = join *=~ (sklansky (dep-1) ~||~ sklansky (dep-1))
  where
    join = (row w1 ~||* w3) -||- (row d2 ~||* d)
      where
        (d,d2,w1,_,w3,_) = params
```

The parameter blocks are taken from the global variable `params`. The local parameter `dep` determines the depth of the circuit. For each recursive call, this parameter is decreased by one. Figure 2.13 shows this structure instantiated for 16 inputs, both for single bits and for pairs of bits. The distinction between the two cases is simply made by choosing parameter blocks from figure 2.12(a) or 2.12(b).

Brent-Kung consists of a sequence of the slices from figure 2.8(b). Each slice contains a forwards and a backwards tree. The forwards tree is:



(a) (b)

Figure 2.13: 16-bit Sklansky for (a) single bits and (b) pairs of bits

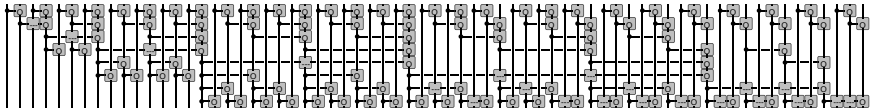


Figure 2.14: Wired layout of the new design

```
fwdTree 1   = thinEmptyX1
fwdTree dep = join *=~ (fwdTree (dep-1) ~||~ fwdTree (dep-1))
  where
    join = (row w1 ~||* w3) -||- (row w2 ~||* d)
      where (d,d2,w1,w2,w3,_) = params
```

Note the similarity to the definition of `sklansky`. Only a parameter to the second occurrence of `row` has changed.

`backTree` shares the same structure, but with extra control to take care of the slide-back (figure 2.8(b)). Brent-Kung is then defined as

```
(d,_,w1,_,w3,w4) = params

bk True  1   = colN 1 (w3 *||* d)
bk _     1   = rowN 2 w4 ~=~ ((w1 *||* w3) *=* (w3 *||* d))
bk first dep = wFill ~=~ bk False (dep-1)
               ~||~ (backTree first True dep ~=~ fwdTree dep)
  where
    wFill = if depth==2 then thinEmptyX
                        else row w4 ~=~ (row w4 ~||* (w3 *=* w3))
```

The recursive case places a smaller Brent-Kung next to a slice consisting of a forwards and backwards tree. The rest of the code is a bit messy due to the fact that the slide-back destroys the regularity around the base case.

The new structure in figure 2.11 is also based on slices, and can be described in Wired without introducing any new ideas. Its Wired layout is shown in figure 2.14.

2.3.3 Results

The parameters used in our estimations are (see also figure 2.5):

T_{int}	R_o	C_g	r	c
50.1ps	23.4kΩ	0.072fF	$0.0229\Omega/\lambda$	$1.43\text{aF}/\lambda$

We analyze for a 100nm process ($\lambda = 50$nm, half the technology feature size), following a similar analysis by Huang and Ercegovac [HE00]. This is not a normal process node, but rather a speculative process derived from NTRS'97 [NTR97]. The gate parameters refer to a min. size gate, but in the analyses, the output resistance is that of a $5\times$ min. size device, while input capacitance is $1.5\times$ min. size. Wiring parameters r and c are obtained by assuming a wire width of 4λ (see formula (4) in [HE00]). The operator cells are square, with a side length of 160λ.

The delays in nanoseconds resulting from the analysis of Sklansky, Brent-Kung and the new structure for 64 bits are (starting from the least significant output):

Sklansky:	0.010, 0.058, 0.10, 0.11, ... 0.42, 0.42, 0.42, 0.42
Brent-Kung:	0.015, 0.066, 0.11, 0.12, ... 0.51, 0.51, 0.55, 0.36
New:	0.012, 0.062, 0.11, 0.11, ... 0.40, 0.44, 0.44, 0.40

The result for Sklansky is very closely in line with those in [HE00], and the new structure has comparable speed despite having fewer operators.

2.4 Implementation

The implementation used in this chapter is complicated and hard to present in its entirety. Here we present some details because they had an important influence on the later development of a simpler system.

To implement the relational circuit model, we do not need the full power of relations as non-deterministic computations; all we need is deterministic computations with abstract information flow. Remember the `Description` type from section 2.1.1:

```
data Description = Primitive Surface Relation
                 | Combined  Combinator Description Description
                 | Generic   Surface (Surface -> Maybe Description)
```

Here, the `Relation` argument to `Primitive` is actually implemented as a function that derives additional information from a partially instantiated surface:

```
type Relation = Surface -> Surface
```

To illustrate this, imagine a system where integers have the additional value `X` for "unknown", and where our surfaces only consist of two integers. Then the binary relation $\{(a, b) \mid b = a + 1\}$ could be modeled by the function

```
incRel (X,X) = (X,X)
incRel (a,X) = (a,a+1)
incRel (X,b) = (b-1,b)
incRel (a,b)
    | b==a+1    = (a,b)
    | otherwise = error "Inconsistency"
```

We see that the operators `+` and `-` are only ever applied to known values, so one way to view this is as a way of lifting normal functions to a domain where the value `X` has been added. We can also view them as "suspended" computations which are only run when enough information is available.

Here is the definition of the wiring primitive `cro` in this model:

```
cro = Primitive (sX 1, sX 1, sX 1, sX 1) croRel
  where
    croRel (pA,pB,pC,pD) = (pA',pB',pC',pD')
      where
        (pA',pD') = connectPorts_sig params (tag (2*scale)) pA pD
        (pB',pC') = connectPorts_sig params (tag (2*scale)) pB pC
```

The surface consists of four s-contacts (`sX`) and the "relation" `croRel` calls a dedicated connection function in order to propagate information between the opposite contacts. The connection function behaves like a wire segment of length `2*scale`.

Generic descriptions (section 2.1.1) are also implemented using a function that looks at the current surface and possibly derives someting new – in this case a whole new sub-description. Once again, we see the correspondence to a suspended computation.

The instantiation engine works by traversing the combinator tree (constructed using the `Combined` constructor) and propagating newly derived information between adjacent sub-blocks until no more information can be derived. Naturally, this fixpoint iteration is quite time consuming; it naively follows the combinator structure rather than the actual data-dependencies, which means that it may repeatedly attempt to run the same relation without gaining any new information. Indeed, running a full timing analysis on a 128-bit circuit takes more than 30 minutes to complete.

2.5 Discussion

This chapter has shown that it is possible to have a hardware description language with simultaneous functional and geometrical meaning even at quite a

low level of abstraction. We also saw that common parallel prefix networks can be described quite concisely in such a system. However, the implementation was shown to be quite inefficient and ad hoc. Is there any hope of improving that?

Yes! The idea of suspending computations when applied to partially unknown data (see section 2.4) is not new. In the functional-logic programming field, the strategy is known as *residuation* [HKMN95]. This was independently spotted by John Hughes and Matthew Naylor, who felt that it should be possible to make a cleaner and better implementation of Wired by building on previous work on residuation. This was what led Naylor to start the work presented in chapter 3. The new implementation based on residuation is presented in chapter 4. Note, however, that this implementation is not used anymore – it is mainly included for historical reasons.

Luckily, the story doesn't end there. Suspended computations (or "thunks") are also used to implement Haskell's lazy evaluation [Jon92]. In fact, it turns out that the signal flow abstraction that was the main motivation for making Wired relational can indeed be obtained using plain old lazy evaluation. The new implementation of Wired, which uses this approach, is presented in part II of the thesis, and the implementation of lazy signal flow abstraction is given in chapter 7.

Chapter 3

A logic programming library in Haskell

This chapter is based on a revised and cut-down version of our Haskell '07 paper [NAR07]. It shows the development of a Haskell library (referred to by the name "LP" in the rest of the thesis) for functional-logic programming, motivated by the needs of Wired. The interested reader is referred to the original paper, which also discusses other applications of the library, including test-data generation, and various extensions, including lazy narrowing. Section 3.6 has been added in order to link the chapter with the rest of the thesis. It should be noted that the LP library is not used in the current implementation of Wired (see part II of the thesis).

3.1 Introduction

We have seen that relations form a very natural basis for describing and modeling digital circuits (sections 1.3.3 and 1.4), but the implementation used in early Wired was found to be unnecessarily complex (chapter 2). A more promising approach to implementing Wired is to embed it in a language that supports both functional and logic programming features. Here, we choose to do this in Haskell and provide the necessary logic programming features in the form of a Haskell library. This lets us keep Wired in a language that is relatively mature compared to dedicated functional-logic languages, such as Curry [HKMN95], and also to integrate Wired with the Lava system (sections 1.3 and 7.2).

Let us illustrate the library with an example. Suppose that we wish to define a list concatenation predicate, `append`. First we must introduce a logical data

type for lists, so we write:

$$(\texttt{nil} ::: (\triangleright)) \;=\; \texttt{datatype}\ (\texttt{cons}_0\ [\,] \curlyvee \texttt{cons}_2\ (:))$$

This definition automatically introduces two list constructors for logical terms with the following types:

$$\begin{array}{lcl} \texttt{nil} & :: & \texttt{Term}\ [a] \\ (\triangleright) & :: & \texttt{Term}\ a \to \texttt{Term}\ [a] \to \texttt{Term}\ [a] \end{array}$$

Comparing to the constructors for ordinary Haskell lists

$$\begin{array}{lcl} [\,] & :: & [a] \\ (:) & :: & a \to [a] \to [a] \end{array}$$

we see that there is a clear type-level correspondence between logical terms, of type $\texttt{Term}\ a$, and normal Haskell values of type a.

Now, to define the `append` predicate using our library, we write

$$\begin{array}{l} \texttt{append} \;::\; \texttt{Term}\ [a] \to \texttt{Term}\ [a] \to \texttt{Term}\ [a] \to \texttt{LP}\ () \\ \texttt{append}\ as\ bs\ cs \;=\; \texttt{caseOf}\ (as, cs)\ \texttt{alts} \\ \quad \textbf{where} \\ \quad\quad \texttt{alts}\ (a, as, cs) \;= \\ \quad\quad\quad\quad (\texttt{nil} \;\;, cs \quad\;) \;\twoheadrightarrow\; (bs\ \texttt{===}\ cs) \\ \quad\quad\quad \oplus\ (a \triangleright as, a \triangleright cs) \;\twoheadrightarrow\; \texttt{append}\ as\ bs\ cs \end{array}$$

Here we make use of a function `caseOf` that allows pattern matching. The pattern variables in the case alternatives `alts` are explicitly quantified by introducing a function for them. The first case says that if as is the empty list, then bs and cs should be equal. In the second case, we require that the first element of as and cs should be the same, and that `append` should hold for their tails.

We can now use this predicate in a computation, for example:

$$\begin{array}{l} \texttt{testApp} \;=\; \textbf{do} \\ \quad (as_{rest}, bs, c_2) \leftarrow \texttt{free} \\ \quad \textbf{let}\ as \;=\; \texttt{int}\ 5 \triangleright \texttt{int}\ 6 \triangleright as_{rest} \\ \quad \textbf{let}\ cs \;=\; \texttt{int}\ 5 \triangleright c_2 \quad\; \triangleright \texttt{int}\ 7 \triangleright \texttt{int}\ 8 \triangleright \texttt{nil} \\ \quad \texttt{append}\ as\ bs\ cs \\ \quad \texttt{return}\ (\texttt{pair}\ as_{rest}\ c_2) \end{array}$$

This says that as is the list $[5, 6, as_{rest}]$ and cs is the list $[5, c_2, 7, 8]$, where as_{rest} and cs are free variables. Note that variables have to be explicitly quantified. In addition we require that the `append` predicate holds for the arguments as, bs and cs, where bs is a free variable. By returning the pair of as_{rest} and c_2, we give the caller of `testApp` access to this term.

This computation can now be tested at the GHCi prompt:

```
*Main> runLP testApp1
[([],6),([7],6),([7,8],6)]
```

We got three possible solutions. In each one, as_{rest} takes up a different portion of the tail of cs, and in all cases, c_2 has the value 6.

The rest of the chapter is divided as follows: Section 3.2 explains some necessary background. Section 3.3 shows the implementation of the logic programming library we just demonstrated. Section 3.4 demonstrates a simple implementation of Wired in the library. The chapter ends with a discussion and a section about an alternative design decision.

3.2 Preliminaries

Logic programs describe *backtracking* computations with *state*, where the state partially maps *logical variables* to values. In a pure functional language such as Haskell, both backtracking and state are computational effects that can be conveniently structured using *monads* [Wad93]. In this section we introduce a backtracking state monad that we will later use as a basis for functional-logic programming in Haskell. Readers familiar with monads and their uses may wish to skip to section 3.3.

3.2.1 Monads

Sometimes the flow of data in a pure functional program is, in Wadler's words, "painfully explicit" [Wad93]. The problem is that the meaning of a program can become "buried under the plumbing required to carry data from its point of creation to its point of use". This plumbing is particularly annoying when the data is frequently accessed and passed on in the same ways, over and over again.

As is often the case when programming, the problem is one of finding the right *abstraction*. Wadler's solution is to use a particularly general abstraction called a *monad*. Whereas a pure computation is, in general, a function of type $a \to b$, a monadic computation is one of type $a \to m\ b$, where m is a monad that captures some *implicit side-effect*. Wadler shows that many of the side-effects found in impure languages, such as state, exceptions and non-determinism, can be simulated using monads.

More specifically, a monad is an abstract data type, parametrised by some other type, that provides the two functions of the following type class:

$$\begin{array}{l}\textbf{class}\ \texttt{Monad}\ m\ \textbf{where} \\ \quad \texttt{return} \ :: \ a \to m\ a \\ \quad (\gg\!=) \ :: \ m\ a \to (a \to m\ b) \to m\ b\end{array}$$

An expression of the form `return` a denotes a computation that simply returns a without any side-effect. And one of the form $m \gg\!= f$ denotes a computation that sequences the two computations m and $f\ a$, where a is the value returned by m.

3.2.2 Monadic exception handling

One kind of side-effect that is useful for some computations to have is *exception handling*. In exception handling, a computation can either return no value, if it *fails* (i.e. raises an exception), or a single value otherwise. Such a computation can be represented using the following data type:

$$\textbf{data}\ \texttt{Maybe}\ a\ =\ \texttt{Nothing} \mid \texttt{Just}\ a$$

To illustrate exception handling, suppose that a computation c, of type `Maybe` a, is sequentially composed of two smaller ones, c_0 and c_1. If c_0 fails then c should fail without ever executing c_1. Otherwise, c should fail if and only if c_1 fails. This behaviour can be captured as a monadic side-effect, freeing the programmer from continuously checking for, and propagating, failure.

$$\begin{array}{l}\textbf{instance}\ \texttt{Monad}\ \texttt{Maybe}\ \textbf{where}\\ \quad \begin{array}{lcl}\texttt{return}\ a & = & \texttt{Just}\ a\\ \texttt{Nothing} \gg\!= f & = & \texttt{Nothing}\\ \texttt{Just}\ a \gg\!= f & = & f\ a\end{array}\end{array}$$

Combining computations in this way can be thought of as "and" combination, because a computation succeeds only if both its constituents do. "Or" combination is also useful. It allows a computation to detect and recover from failure. A general interface to "or" combination of monadic computations is provided by the following type class:

$$\begin{array}{l}\textbf{class}\ \texttt{Monad}\ m \Rightarrow \texttt{MonadPlus}\ m\ \textbf{where}\\ \quad \begin{array}{lcl}\texttt{mzero} & :: & m\ a\\ (\oplus) & :: & m\ a \to m\ a \to m\ a\end{array}\end{array}$$

In exception handling, `mzero` denotes a computation that fails, and $c_0 \oplus c_1$ denotes a computation that executes c_0, and if that fails, then c_1.

$$\begin{array}{l}\textbf{instance}\ \texttt{MonadPlus}\ \texttt{Maybe}\ \textbf{where}\\ \quad \begin{array}{lcl}\texttt{mzero} & = & \texttt{Nothing}\\ \texttt{Nothing} \oplus m & = & m\\ \texttt{Just}\ a \oplus m & = & \texttt{Just}\ a\end{array}\end{array}$$

Example 1 (*Failing lookups*). Consider a function, `lookup`, that takes a key, and a list of key/value pairs, and returns the value that is paired with the given key. If the given key is not paired with any value in the list, then `lookup` should fail. With the help of two useful abstractions,

$$\begin{array}{lcl} m_0 \gg m_1 & = & m_0 \gg\!= (\lambda\_ \to m_1) \\ \texttt{guard}\ c & = & \textbf{if}\ \ c\ \ \textbf{then}\ \ \texttt{return}\ () \\ & & \quad\ \textbf{else}\ \ \texttt{mzero} \end{array}$$

the `lookup` function can be defined as follows:

$$\begin{array}{lcl} \texttt{lookup}\ x\ [\,] & = & \texttt{mzero} \\ \texttt{lookup}\ x\ ((k,v) : ps) & = & (\texttt{guard}\ (x == k) \gg \texttt{return}\ v) \\ & & \quad \oplus \\ & & \quad \texttt{lookup}\ x\ ps \end{array}$$

If the given key is paired with more than one element in the list, then the value of the *first* matching pair is returned. For example, the following sample evaluations hold:

$$\begin{array}{lcl} \texttt{lookup 'a'}\ [(\texttt{'a'},1),(\texttt{'b'},3),(\texttt{'a'},6)] & \rightsquigarrow & \texttt{Just}\ 1 \\ \texttt{lookup 'c'}\ [(\texttt{'a'},1),(\texttt{'b'},3),(\texttt{'a'},6)] & \rightsquigarrow & \texttt{Nothing} \end{array}$$

Using the exception handling monad, it is straightforward to define a function that performs two lookups and adds the results.

$$\begin{array}{l} \texttt{add}\ k_0\ k_1\ l\ \ = \\ \quad \texttt{lookup}\ k_0\ l \gg\!= \lambda a \to \\ \quad \texttt{lookup}\ k_1\ l \gg\!= \lambda b \to \\ \quad \texttt{return}\ (a{+}b) \end{array} \qquad \begin{array}{l} \texttt{add}\ k_0\ k_1\ l\ \ = \ \ \textbf{do} \\ \quad a \leftarrow \texttt{lookup}\ k_0\ l \\ \quad b \leftarrow \texttt{lookup}\ k_1\ l \\ \quad \texttt{return}\ (a{+}b) \end{array}$$

The two definitions are equivalent. The one on the right simply makes use of syntactic sugar for monads, known as *do-notation*.

The possibility that the first lookup may fail does not need to be considered explicitly by the second, as failure is propagated implicitly as a monadic side-effect. For example, the following evaluations hold:

$$\begin{array}{lcl} \texttt{add 'a' 'b'}\ [(\texttt{'a'},1),(\texttt{'b'},3),(\texttt{'a'},6)] & \rightsquigarrow & \texttt{Just}\ 4 \\ \texttt{add 'c' 'a'}\ [(\texttt{'a'},1),(\texttt{'b'},3),(\texttt{'a'},6)] & \rightsquigarrow & \texttt{Nothing} \end{array}$$

□

3.2.3 Monadic backtracking

A natural generalisation of exception handling is *backtracking*. In backtracking, a computation can yield zero or more results, not just zero or one. Therefore, when a computation fails, it may be possible to backtrack to an earlier computation, pick a different result, and then try executing the original computation again. Haskell's list monad provides such behaviour.

$$
\begin{array}{l}
\textbf{instance}\ \texttt{Monad}\ [\,]\ \textbf{where} \\
\quad \texttt{return}\ a \quad = \quad [a] \\
\quad [\,] \quad \gg\!= f \quad = \quad [\,] \\
\quad (a : as) \gg\!= f \quad = \quad f\ a +\!\!+ (as \gg\!= f)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \texttt{MonadPlus}\ [\,]\ \textbf{where} \\
\quad \texttt{mzero} \quad = \quad [\,] \\
\quad (\oplus) \quad = \quad (+\!\!+)
\end{array}
$$

Example 2 (*Backtracking lookups*). In exception handling, the `lookup` function returns only the first value that is paired with the given key in the list. In backtracking, all associated values are returned as a lazily evaluated list.

$$
\begin{array}{lcl}
\texttt{lookup 'a'}\ [(\texttt{'a'}, 1), (\texttt{'b'}, 3), (\texttt{'a'}, 6)] & \rightsquigarrow & [1, 6] \\
\texttt{lookup 'c'}\ [(\texttt{'a'}, 1), (\texttt{'b'}, 3), (\texttt{'a'}, 6)] & \rightsquigarrow & [\,]
\end{array}
$$

Note that the definition of `lookup` has not changed. The `add` function now returns the results of all combinations of lookups.

$$
\begin{array}{lcl}
\texttt{add 'a' 'a'}\ [(\texttt{'a'}, 1), (\texttt{'b'}, 3), (\texttt{'a'}, 6)] & \rightsquigarrow & [2, 7, 7, 12] \\
\texttt{add 'a' 'c'}\ [(\texttt{'a'}, 1), (\texttt{'b'}, 3), (\texttt{'a'}, 6)] & \rightsquigarrow & [\,]
\end{array}
$$

□

3.2.4 Adding state

Another kind of side-effect that is useful for some computations to have is *state passing*, whereby state is implicitly threaded through a sequence of computations, and each individual computation can read, modify or ignore it. A state passing computation can be represented as a transition function from the current state to a pair containing the next state and a return value. To support both state *and* backtracking, we use a transition function to a *list* of such pairs.

$$
\textbf{newtype}\ \texttt{BS}\ s\ a \quad = \quad \texttt{BS}\ \{\ \texttt{run} \quad :: \quad s \rightarrow [(a, s)]\ \}
$$

Here BS stands for *backtracking state*. For any type of state s, BS s can be made a monad as follows:

$$\begin{array}{l}\textbf{instance}\ \texttt{Monad}\ (\texttt{BS}\ s)\ \textbf{where}\\ \quad \texttt{return}\ a \;=\; \texttt{BS}\ (\lambda s \to [(a, s)])\\ \quad m \gg\!= f \;=\; \texttt{BS}\ (\lambda s \to \texttt{run}\ m\ s \gg\!= (\lambda(a, s) \to \texttt{run}\ (f\ a)\ s))\end{array}$$

The occurrence of $(\gg\!=)$ on the right-hand side of the second equation refers to the $(\gg\!=)$ of the list monad instance. So "and" combination passes the current state to the first computation which yields a list of next-states, each of which is passed in turn to the second computation. "Or" combination passes the same current state to each alternative.

$$\begin{array}{l}\textbf{instance}\ \texttt{MonadPlus}\ (\texttt{BS}\ s)\ \textbf{where}\\ \quad \texttt{mzero} \;=\; \texttt{BS}\ (\lambda s \to [\,])\\ \quad m_0 \oplus m_1 \;=\; \texttt{BS}\ (\lambda s \to \texttt{run}\ m_0\ s +\!\!+ \texttt{run}\ m_1\ s)\end{array}$$

To abstract away from the internal representation of the BS monad, we define a get computation that returns the current state, and a put computation that replaces the current state with the given state.

$$\begin{array}{lcl}\texttt{get} & :: & \texttt{BS}\ s\ s\\ \texttt{get} & = & \texttt{BS}\ (\lambda s \to [(s, s)])\\ \texttt{put} & :: & s \to \texttt{BS}\ s\ ()\\ \texttt{put}\ s & = & \texttt{BS}\ (\lambda\_ \to [((), s)])\end{array}$$

We chose to present this backtracking state monad here because of its simplicity, but in practice we use the more efficient variant, shown in figure 3.14, based on Hinze's two-continuation monad transformer [Hin00].

3.2.5 Adding variables

The implicit state of a logic program is a partial mapping from logical variables to values, often referred to as the current *substitution*. We specialise the backtracking state monad to support such a substitution as follows:

$$\begin{array}{lcl}\textbf{type}\ \texttt{BV}\ v\ a & = & \texttt{BS}\ (\texttt{IntMap}\ v, \texttt{VarID})\ a\\ \textbf{type}\ \texttt{VarID} & = & \texttt{Int}\end{array}$$

The state of the BS monad is hard-coded to be a pair containing the substitution and its size. Variable identifiers are represented as integers, so the substitution is a mapping (IntMap) from integers to values of any type v. An efficient implementation of the IntMap data structure is provided in Haskell's hierarchical

```
newVar       :: a → BV a VarID
newVar a     =  do (env, i) ← get
                   put (insert i a env, i+1)
                   return i

readVar      :: VarID → BV a a
readVar v    =  do (env, i) ← get
                   return (env ! v)

writeVar     :: VarID → a → BV a ()
writeVar v a =  do (env, i) ← get
                   put (insert v a env, i)
```

Figure 3.1: Operations of the "backtracking variables" monad

libraries, but for completeness we specify the functionality that we expect of it in figure 3.15. Functions to create, read, and write variables are defined in figure 3.1.

Claessen describes an alternative way to implement logical variables using the support for *mutable references* provided by Haskell's `ST` monad [CL00]. The advantage of using `ST` references is that they are polymorphic, automatically garbage collected, and can be accessed in constant time. Indeed, we believe that Claessen's approach will outperform ours, but here we have chosen to present a simple implementation that performs reasonably well in practice.

3.3 A library for logic programming

In this section we build a layer of logic programming support on top of the `BV` monad, including functions for creating *logical data types*, creating *logical variables*, *unification*, *residuation*, and *pattern matching*. We demonstrate the resulting library by describing some useful arithmetic and list-based relations.

3.3.1 Logical terms

A *logical term* is a value, similar to the value of an algebraic data type in Haskell, that can be constructed using one of a number of data constructors. However, every logical term has the additional possibility of being a *logical variable*. For example, a logical list term could be constructed by a "nil" constructor, a "cons" constructor *or a variable constructor*. Because such data types may be recursively defined, it is not, in general, possible to treat the values of existing Haskell

data types as logical terms.

Instead, we define a *universal* data type with a specific constructor for variables, in which any algebraic data type can be encoded.

```
data Uni = Var VarID | Ctr Int [Uni] | Int Int
```

The universal data type `Uni` provides constructors for logical variables, compound terms, and primitive Haskell data types. (For presentation purposes, we support only one primitive type, namely `Int`.)

3.3.2 Logical variables

The value of a logical variable is either *bound* to a logical term or *unbound*, so it can be represented using Haskell's `Maybe` type.

```
type Val = Maybe Uni
```

A monad for logic programming, `LP`, can be defined by hard coding the values of variables to be of type `Val`.

```
type LP a = BV Val a
```

`VarID` is an abstract data type that provides the following three operations, as defined in figure 3.2: `unboundVar` for creating a new, unbound, variable; `bindVar` for binding a value to an unbound variable; and `ifBound` for calling one of two given functions depending on whether a given variable is bound or not. We will be able to use the same interface when `Val` is later redefined to support residuation.

3.3.3 Unification

Two logical terms can be successfully *unified* if they are equal, or if they contain unbound variables that can be instantiated so as to make them equal. However, to obtain maximum generality, unification must instantiate as *few* variables as possible. This can be achieved by allowing unbound variables to be unified with each other without having to instantiate them.

Two unbound variables can be unified simply by letting one be bound to the other. Since a variable will never be bound twice, a set of bindings like $[a \mapsto b, b \mapsto c, d \mapsto b, e \mapsto f]$ forms a set of trees, and no variable will appear in two trees. We take the variables at the roots of the trees as the representatives of the equivalence sets. Cycles can be easily avoided when binding one variable

```
unboundVar    :: LP VarID
unboundVar    =  newVar Nothing

bindVar       :: VarID → Uni → LP ()
bindVar v a   =  writeVar v (Just a)

ifBound       :: VarID → (Uni → LP b) → LP b → LP b
ifBound v t f =  readVar v ≫= decons
  where
    decons (Just a)  =  t a
    decons Nothing   =  f
```

Figure 3.2: An implementation of logical variables

to another by checking that the corresponding trees do not have the same root. An important operation in the unification algorithm is therefore to find the root of a logical term:

```
root          :: Uni → LP Uni
root (Var v)  =  ifBound v root (return (Var v))
root a        =  return a
```

The unification function takes two terms that are to be unified and performs a case analysis on their roots, as shown in figure 3.3. If both roots are the same variable, then unification succeeds. If at least one of the roots is a variable, then that variable is bound to the other root. If both roots are instantiated to the same term constructor, then the arguments of those constructors are unified. In any other case, unification fails.

3.3.4 Static typing

A problem with the universal data type representation is that every logical term has the *same* type, namely `Uni`. Leijen and Meijer propose *phantom types* as an elegant solution to this problem [LM99]. Their idea is to create a data type with a type parameter that does no occur in any construction. This type parameter is referred to as a phantom type.

```
newtype Term a  =  Term { uni :: Uni }
```

Now terms have the type `Term` a for some type a that can be controlled by an explicit type signature. This explicit typing allows data constructors for terms

to be defined with the desired type. For example, the following functions define the list constructors:

```
nil     ::  Term [a]
nil     =   Term (Ctr 0 [])
(▷)     ::  Term a → Term [a] → Term [a]
a ▷ b   =   Term (Ctr 1 [uni a, uni b])
```

Only well-typed logical terms can be constructed using these functions. It is slightly awkward that each constructor must *manually* be given a type signature, a unique identifier, and a representation in the universal type. Thankfully, these three tasks can be completely automated. Using the combinators of figure 3.4, the list constructors are simply defined as

```
(nil ::: (▷))  =  datatype (cons0 [] ϒ cons2 (:))
```

and constructurs for a few other basic types are defined similarly in figure 3.5. It is straightforward to support any algebraic type in this way. However, primitive types such as `Int` must be defined specially:

```
int     ::  Int → Term Int
int n   =   Term (Int n)
```

3.3.5 Logical interface

It is useful to overload unification and free-variable creation so that they can be used on a number of different types. Therefore, we introduce the following type class for logical terms:

```
class Logical a where
  free    ::  LP a
  (===)   ::  a → a → LP ()
  match   ::  Logical b ⇒ a → a → LP b → LP b
```

For now, the `match` member is not important – it will be used in the implementation of residuation in Section 3.3.8. To instantiate `Term` a for all a under the `Logical` class, we just need to convert between the typed and universal representations.

```
instance Logical (Term a) where
  free          =  do v ← unboundVar
                      return (Term (Var v))
  a === b       =  unify (uni a) (uni b)
  match a b k   =  match′ (uni a) (uni b) k
```

```
unify       ::  Uni → Uni → LP ()
unify a b   =   do  a_r ← root a ; b_r ← root b ; unif a_r b_r
  where
    unif (Var v)    (Var w)     | v == w   =  return ()
    unif (Var v)    b                     =  bindVar v b
    unif a          (Var w)               =  bindVar w a
    unif (Int a)    (Int b)     | a == b   =  return ()
    unif (Ctr n as) (Ctr m bs)  | n == m   =  unif' as bs
    unif _          _                     =  mzero
    unif' []        []          =  return ()
    unif' (a : as)  (b : bs)    =  unify a b ≫ unif' as bs
```

Figure 3.3: Unification algorithm

```
cons0          ::  a → Int → Term a
cons0 f        =   λn → Term (Ctr n [])
cons1          ::  (a → b) → Int → Term a → Term b
cons1 f        =   λn a → Term (Ctr n [uni a])
cons2          ::  (a → b → c) → Int → Term a → Term b → Term c
cons2 f        =   λn a b → Term (Ctr n [uni a, uni b])

data Pair a b  =   a ::: b
a ϒ b          =   λn → a n ::: b (n+1)
datatype d     =   d 0
```

Figure 3.4: Combinators for creating logical data types

```
(nothing ::: just)  =  datatype (cons0 Nothing ϒ cons1 Just)
(false   ::: true)  =  datatype (cons0 False   ϒ cons0 True)
pair                =  datatype (cons2 (,))
```

Figure 3.5: Constructors for basic types

The main motivation for having the `Logical` class is to support *tuple-terms*. Tuple-terms allow many variables to be created in one go, and collections of terms to be treated as if they were a single compound term. For example, the following instance defines tuple-terms of size two:

$$
\begin{array}{l}
\textbf{instance}\ (\texttt{Logical}\ a, \texttt{Logical}\ b) \Rightarrow \texttt{Logical}\ (a,b)\ \textbf{where} \\
\quad \texttt{free} \ = \ \textbf{do}\ a \leftarrow \texttt{free}\ ;\ b \leftarrow \texttt{free} \\
\qquad\qquad\quad \texttt{return}\ (a,b) \\
\quad (a_0,a_1)\ \texttt{===}\ (b_0,b_1) \quad = \quad a_0\ \texttt{===}\ b_0 \gg a_1\ \texttt{===}\ b_1 \\
\quad \texttt{match}\ (a_0,a_1)\ (b_0,b_1)\ k \ = \ \texttt{match}\ a_0\ b_0\ (\texttt{match}\ a_1\ b_1\ k)
\end{array}
$$

Tuple-terms of any size can be defined similarly.

Example 3 (*List concatenation*). Given three lists, as, bs, and cs – that may be variables or contain variables – the list concatenation relation non-deterministically generates all variable instantiations such that cs is the concatenation of as and bs.

$$
\begin{array}{lcl}
\texttt{app} & :: & \texttt{Term}\ [a] \rightarrow \texttt{Term}\ [a] \rightarrow \texttt{Term}\ [a] \rightarrow \texttt{LP}\ () \\
\texttt{app}\ as\ bs\ cs & = & \textbf{do}\ \ as\ \texttt{===}\ \texttt{nil}\ ;\ bs\ \texttt{===}\ cs \\
 & \oplus & \textbf{do}\ \ (a, as', cs') \leftarrow \texttt{free} \\
 & & \qquad as\ \texttt{===}\ (a \triangleright as') \\
 & & \qquad cs\ \texttt{===}\ (a \triangleright cs') \\
 & & \qquad \texttt{app}\ as'\ bs\ cs'
\end{array}
$$

This definition of `app` is said to be *polymodal*, meaning it has "many modes" of operation. For example, if any two of as, bs and cs are known then the other can be inferred. □

3.3.6 Pattern matching

Term deconstruction is achieved using free variables and unification, as illustrated by the above example. However, it is often clearer and more concise to use pattern matching to deconstruct terms. Although we cannot use Haskell's **case** construct for this purpose, we can redefine `app` as follows:

$$
\begin{array}{l}
\texttt{app}\ as\ bs\ cs \ = \ \texttt{caseOf}\ (as, cs)\ \texttt{alts} \\
\quad \textbf{where} \\
\qquad \texttt{alts}\ (a, as, cs) \ = \\
\qquad\qquad (\texttt{nil}\ \ , cs\ \ \ ) \twoheadrightarrow (bs\ \texttt{===}\ cs) \\
\qquad\quad \oplus\ (a \triangleright as, a \triangleright cs) \twoheadrightarrow \texttt{app}\ as\ bs\ cs
\end{array}
$$

The `caseOf` function can be thought of as a replacement for Haskell's **case** construct in which logical terms can be matched. The variable `alts` stands for

"case alternatives" and is parameterised by the free variables that appear in the patterns of all the alternatives.

Now it remains to define `caseOf` and the $\twoheadrightarrow$ operator.

$$
\begin{array}{lcl}
pat \twoheadrightarrow rhs & = & \texttt{return}\ (pat, rhs) \\
\texttt{caseOf}\ a\ as & = & \textbf{do}\ \ (pat, rhs) \leftarrow \texttt{free} \gg\!= as \\
 & & \quad\ pat\ \texttt{===}\ a \\
 & & \quad\ rhs
\end{array}
$$

3.3.7 Residuation

When the value of an uninstantiated logical variable is needed for a computation to proceed, one strategy, demonstrated above, is to non-deterministically instantiate it with appropriate values. An alternative strategy, known as *residuation*, is to *suspend* the computation until the variable becomes bound, and to proceed by evaluating the next, sequentially composed, computation. The suspended computation (also called *residual*) is *resumed* as soon as the variable that it is waiting on is instantiated.

We implement residuation by associating every unbound variable with a list of computations that are suspended on it. To do this, we need to alter our model of logical variables, and redefine the `Val` type as follows:

$$
\begin{array}{lcl}
\textbf{data}\ \texttt{Val} & = & \texttt{Bound}\ \texttt{Uni}\ \mid\ \texttt{Unbound}\ [\texttt{Residual}] \\
\textbf{type}\ \texttt{Residual} & = & \texttt{Uni} \rightarrow \texttt{LP}\ ()
\end{array}
$$

A logical variable is now either bound to a value, or unbound, with a list of residuals waiting for it to become bound.

The new implementation of the `VarID` abstract data type is shown in figure 3.6. The important difference is in the definition of `bindVar`: when an unbound variable becomes bound, each of its associated residuals is resumed by calling the `resumeOn` function, as defined in figure 3.7. An expression of the form rs `` `resumeOn` `` v applies each residual in rs to the value v, provided that v is instantiated. If it is not, then v must be an unbound variable, so the list of residuals associated with v is updated to include rs.

To create a function that residuates, we introduce a special application operator called `rigid'`:

```
unboundVar    =  newVar (Unbound [])

bindVar v a   =  do Unbound rs ← readVar v
                    writeVar v (Bound a)
                    rs `resumeOn` a

ifBound v f g =  readVar v >>= decons
  where
    decons (Bound a)   =  f a
    decons (Unbound rs) =  g
```

Figure 3.6: An implementation of logical variables that supports residuation

```
resumeOn              ::  [Residual] → Uni → LP ()
resumeOn rs (Var v)   =   do Unbound ss ← readVar v
                             writeVar v (Unbound (rs ++ ss))
resumeOn rs a         =   mapM_ (resume a) rs
  where
    resume a g  =  g a
```

Figure 3.7: Resuming a list of residuals

```
rigid'      :: Logical b ⇒ (Uni → LP b) → (Uni → LP b)
rigid' f a  =  do a_r ← root a
                  b   ← free
                  let g x  =  do c ← f x
                                 b === c
                  [g] `resumeOn` a_r
                  return b
```

It takes a logical function as argument and returns a new version of it, which automatically suspends when applied to an unbound variable. If a function suspends, we still want to have access to its (future) result. In `rigid'`, the result is represented by the variable b, so we define the residual g to unify the result of f with b. This way, whenever g is resumed, the value of b is immediately updated.

The typed version of `rigid'` is `rigid`, and is defined as:

```
rigid      :: Logical b ⇒ (Term a → LP b) → (Term a → LP b)
rigid f a  =  rigid' (f . Term) (uni a)
```

Example 4 (*Rigid arithmetic*). In order to define efficient deterministic operations on integers, we provide the following primitive deconstructor for integers that can only be defined by inspecting the internal structure of terms:

```
unint      :: Logical b ⇒ Term Int → (Int → LP b) → LP b
unint a f  =  rigid' (λ(Int a) → f a) (uni a)
```

With the help of the following shorthand,

```
liftInt2 f a b  =  unint a (λa →
                     unint b (λb →
                       return (int (f a b))))
```

we can, for example, define rigid functions for addition and subtraction.

```
a +̄ b  =  liftInt2 (+) a b
a −̄ b  =  liftInt2 (−) a b
```

The results of these functions will only become known once their arguments, a and b, have been instantiated. □

Example 5 (*Relational arithmetic*). Using the following generally-useful shorthand,

```
match'        ::  Logical a ⇒ Uni → Uni → LP a → LP a
match' a b k  =   do  a_r ← root a ; b_r ← root b ; ma a_r b_r
  where
    ma (Var v)    (Var w)     | v == w   =  k
    ma (Var v)    b                     =  bindVar v b ≫ k
    ma _          (Var w)               =  rigidMatch
    ma (Int a)    (Int b)     | a == b   =  k
    ma (Ctr n as) (Ctr m bs)  | n == m   =  zipm as bs
    ma _          _                     =  mzero

    zipm []       []          =  k
    zipm (x : xs) (y : ys)    =  match' x y (zipm xs ys)

    rigidMatch  =  rigid' (λb → match' a b k) b
```

Figure 3.8: Rigid matching algorithm

$$a \texttt{ <== } m \;=\; \textbf{do}\;\; b \leftarrow m\,;\; a \texttt{ === } b$$

a deterministic addition relation can be defined as follows:

$$\begin{aligned} a \mathbin{\hat{+}} b \;=\; \textbf{do}\;\; & c \;\leftarrow\; a \mathbin{\bar{+}} b \\ & b \texttt{ <== } c \mathbin{\bar{-}} a \\ & a \texttt{ <== } c \mathbin{\bar{-}} b \\ & \texttt{return } c \end{aligned}$$

Once any two of a, b, and c are known, the other can be inferred. □

3.3.8 Rigid pattern matching

Recall that the `caseOf` function allows term deconstruction via pattern matching. A slight variant of this is `rigidCaseOf`, whereby matching *suspends* until the scrutinee is instantiated at least as much as the pattern it is being matched against. To implement `rigidCaseOf` we use the `match` method of the `Logical` class. An action of the form `match` $p\ a\ k$ executes k if the pattern p matches the scrutinee a. The `rigidCaseOf` function differs from `caseOf` in that the `match` function is used in place of unification.

$$\begin{aligned} \texttt{rigidCaseOf}\ a\ as \;=\; \textbf{do}\;\; & (pat, rhs) \leftarrow \texttt{free} \gg\!= as \\ & \texttt{match}\ pat\ a\ rhs \end{aligned}$$

```
deref a  =  do a_r ← root a
               case a_r of
                    Ctr n as → liftM (Ctr n) (mapM deref as)
                    a        → return a

untermDeref    :: Conv a ⇒ Term a → LP a
untermDeref a  =  do a' ← liftM Term (deref (uni a))
                     unterm a'
```

Figure 3.9: Dereferencing version of `unterm`

Recall that for values of type `Term` a, `match` is defined simply as `match'`. The definition of match' is given in figure 3.8. The main difference from unification is that the continuation k is called when matching succeeds. When an unbound variable in the scrutinee is matched against a bound value in a pattern, then matching is suspended, via a call to `rigid'`, until that variable becomes bound.

Example 6 (*Rigid list concatenation*). A rigid list concatenation relation can be defined as follows:

```
(⧺̄)      :: Term [a] → Term [a] → LP (Term [a])
as ⧺̄ bs   =  rigidCaseOf as alts
  where
    alts (a, as)  =
         nil      ↠ return bs
      ⊕ a ▷ as    ↠ liftM (a ▷) (as ⧺̄ bs)
```

The result is returned once the spine of as is fully instantiated. Although deterministic, the relation permits the values in the lists involved to flow in any direction, for example, from as and bs to the result, and vice-versa. □

3.3.9 Running the computations

In order to run the logical computations, we first need to be able to convert terms to normal values. For this, we introduce the type class `Conv`:

```
class Conv a where
  unterm  :: Term a → LP a
```

Figure 3.16 shows the class instantiated for a few basic types. However, `unterm` doesn't take into account the fact that the term may be a variable. Therefore, we define the function `untermDeref` according to figure 3.9. The helper function `deref` is used to recursively trace the roots of any variables in the term.

Now we can define the run function as

$$\begin{array}{lcl} \texttt{runLP} & :: & \texttt{Conv}\ a \Rightarrow \texttt{LP}\ (\texttt{Term}\ a) \rightarrow [a] \\ \texttt{runLP}\ m & = & \texttt{map}\ \texttt{fst}\ (\texttt{run}\ (m \gg\!= \texttt{untermDeref})\ (\texttt{empty}, 0)) \end{array}$$

That is, we first convert the result from the computation m, and then run the `LP` monad.

3.4 Application to Wired

In this section we implement a simple version of Wired called *MiniWired* in order to demonstrate the benefits of building on a general logic programming framework. We also discuss some alternative design decisions that are made possible by the availability of non-determinism, and demonstrate MiniWired by describing and analysing a parallel prefix circuit.

3.4.1 Descriptions in Wired

In MiniWired, circuits are modelled as rectangular *tiles* that can be composed with each other to give larger tiles. Data may enter or leave a tile on each of its four *edges*, which are named west, north, south and east:

$$\begin{array}{lcl} \textbf{data}\ \texttt{Tile}\ a & = & \texttt{Tile}\ \{\ \texttt{west} \ :: \ \texttt{Edge}\ a \\ & & \ ,\ \texttt{north} \ :: \ \texttt{Edge}\ a \\ & & \ ,\ \texttt{south} \ :: \ \texttt{Edge}\ a \\ & & \ ,\ \texttt{east} \ :: \ \texttt{Edge}\ a \\ & & \ \} \end{array}$$

The *edge* of a tile is a list of *segments*, and each segment may or may not carry a signal[1]. The length of the edge is stored explicitly for convenience, although this information is redundant.

$$\begin{array}{lcl} \textbf{data}\ \texttt{Edge}\ a & = & \texttt{Edge}\ \{\ \texttt{len} \ :: \ \texttt{Term Int} \\ & & \ ,\ \texttt{segs} \ :: \ \texttt{Term}\ [\texttt{Maybe}\ a] \\ & & \ \} \end{array}$$

The following function for constructing an edge from a list of segments is useful:

$$\texttt{edge}\ as \ = \ \texttt{Edge}\ (\texttt{int}\ (\texttt{length}\ as))\ (\texttt{foldr}\ (\triangleright)\ \texttt{nil}\ as)$$

The `Edge` and `Tile` types are isomorphic to pairs and 4-tuples respectively, and can be instantiated under the `Logical` class in the same way that tuples are,

[1] Comparing to section 2.1.1, `Nothing` corresponds to an i-contact and `Just` to an s-contact.

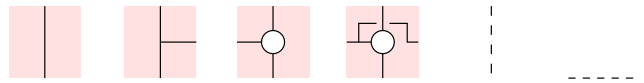


Figure 3.10: Some primitive tiles

```
dot g  =  do (a, b)  ←  free
             c       ←  g a b
             let w   =  edge [just a]
             let n   =  edge [just b]
             let s   =  edge [just c]
             let e   =  edge [nothing]
             return (Tile w n s e)

thinY  =  do we      ←  free
             let ns  =  edge []
             return (Tile we ns ns we)
```

Figure 3.11: Definitions of the primitive tiles `dot` and `thinY`

as described in section 3.3.5. However, tiles are treated slightly differently from 4-tuples in that when they are created, they are constrained to be *rectangular*. The full instance definitions of `Edge` and `Tile` are shown in figure 3.17.

A circuit description is defined to be a relation over tiles. The type variable a represents the type of data that flows through the circuit.

```
type Desc a  =  LP (Tile a)
```

A description has both a physical size and a behaviour. The size is defined by the number of segments in the north and west edges of the tile (or equivalently, the south and east edges). The behaviour is defined by a *relation* between the signals on the edges of the tile. The use of a relation rather than a function greatly reduces the number of distinct primitive tiles that are required. Diagrams of a selection of primitive tiles are shown in figure 3.10. Two of these primitives, `dot` and `thinY` are defined in figure 3.11, and the rest are defined in figure 3.18. Note that the behavior of the "dot" cell is given as the parameter g to `dot` and `crodot`, and the behavior of forking wires is the parameter f to `crodot` and `wireT3`. This is just a more explicit way of achieving what was done as a type-level parameterization of the signal interpretation in section 2.1.3.

```
(*||*)   ::  Desc a → Desc a → Desc a
d0 *||* d1  =  do  t0 ← d0
                   t1 ← d1
                   n  ← join (north t0) (north t1)
                   s  ← join (south t0) (south t1)
                   east t0 === west t1
                   return (Tile (west t0) n s (east t1))
```

Figure 3.12: The "beside" combinator

3.4.2 Combinators

MiniWired allows two circuit descriptions to be combined by placing them one *beside* the other, or one *below* the other. To define these two combinators it is necessary to be able to join the edges of tiles together.

```
join e0 e1  =  liftM2 Edge
                  (len  e0 +̂ len  e1)
                  (segs e0 +̄+ segs e1)
```

Notice that where two edges are joined, their combined length is computed using the addition relation defined in Example 5. The reason for using a relation is to allow the length of an edge, which may be unknown, to be inferred from its surrounding context. For example, the height of `thinY`, which is unconstrained, can be inferred if it is placed beside a circuit of known height, such as `wireY1`. This ability to infer lengths from contexts can simplify circuit description by reducing the amount of information that needs to be given explicitly by the programmer.

The definition of the "beside" combinator is given in figure 3.12. A description is placed beside another by joining the northern and southern edges and unifying the eastern western edges. The "below" combinator is defined similarly in figure 3.19.

A row of tiles can be obtained by iteratively applying the "beside" combinator the desired number of times.

```
rowN 0 d  =  thinY
rowN n d  =  d *||* rowN (n−1) d
```

Sometimes the desired length of a row is obvious from the context in which it is placed, and shouldn't need to be stated explicitly. Such a row – any number of tiles long – can be defined as follows:

$$\texttt{row}\ d \;=\; \texttt{thinY} \oplus (d\ \texttt{*||*}\ \texttt{row}\ d)$$

However, there is a slight problem with this definition: it has infinitely many solutions, and so circuit generation will get stuck down an infinitely long search path. A much more sensible approach is to wait until the length of the surrounding context is known before searching; this way circuit generation should be terminating.

$$\begin{array}{lll} \texttt{row}\ d & = & \textbf{do}\ n \leftarrow \texttt{free} \\ & & \quad\ s \leftarrow \texttt{unint}\ n\ (\lambda n \rightarrow \texttt{thinY} \oplus (d\ \texttt{*||*}\ \texttt{row}\ d)) \\ & & \quad\ n\ \texttt{===}\ \texttt{len}\ (\texttt{north}\ s) \\ & & \quad\ \texttt{return}\ s \end{array}$$

So far, tile edges have not been constrained to have non-negative length, and so by creating negatively sized tiles, `row` may still be non-terminating. This can be resolved by modifying the rigid subtraction operator to fail if its result is negative, as shown in figure 3.20. Alternatively, `row` can be defined to be fully deterministic, as shown in figure 3.21. The deterministic and non-deterministic variations of `row` behave differently when rows are placed beside rows: the former favours expanding the first row, whereas the latter fairly expands each in a non-deterministic fashion. Future experiments with Wired should help us decide which is preferable, but the availability of non-determinism certainly gives the library a more flexible feel.

Example 7 (*Sklansky*). In section 2.3.2, we described the Sklansky parallel prefix network in Wired. Now we can also give a definition in MiniWired:

$$\begin{array}{lcl} \texttt{sklansky}\ f\ g\ 0 & = & \texttt{thinX1} \\ \texttt{sklansky}\ f\ g\ n & = & (\texttt{left}\ \texttt{*=*}\ \texttt{skl})\ \texttt{*||*}\ (\texttt{right}\ \texttt{*=*}\ \texttt{skl}) \\ \quad \textbf{where} & & \\ \quad\quad \texttt{skl} & = & \texttt{sklansky}\ f\ g\ (n{-}1) \\ \quad\quad \texttt{left} & = & \texttt{row}\ \texttt{wireY1}\ \ \ \texttt{*||*}\ \texttt{wireT3}\ f \\ \quad\quad \texttt{right} & = & \texttt{row}\ (\texttt{crodot}\ f\ g)\ \texttt{*||*}\ \texttt{dot}\ g \end{array}$$

Note how the description is parameterized on the signal interpretations f and g. The diagram of an 8-bit instance of this network is shown in figure 3.13. Since there is no reprereresentation of layout in MiniWired (for reasons of simplicity), this picture was drawn manually. The full Hardwired implementation, in section 4.2, has a representation of layout, and this allows the pictures to be generated automatically. □

3.4.3 Analysis

In section 2.2.2, we outlined an RC-delay analysis for Wired circuits by means of non-standard interpretation; however, the actual implementation was too com-

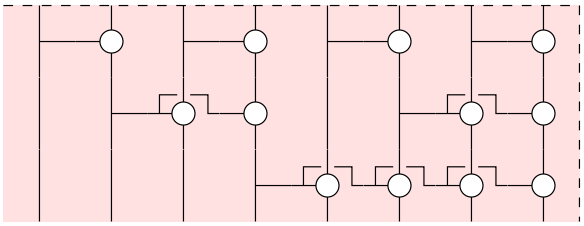


Figure 3.13: The 8-bit instance of Sklansky

plicated to present. Now we are ready to define such an analysis in MiniWired, but for simplicity, we will concentrate on a similar but simpler analysis. The output delay of each gate, d_{out}, is calculated as follows:

$$d_{out} \quad = \quad maximum(d_0, d_1, \cdots, d_n) + d_{int} + (c \times N)$$

Here $d_0 \cdots d_n$ are input delays, d_{int} is the internal delay of the gate, N is the number of gates driven by the output signal and c is a constant which decides the significance of a high fanout. The fanout, N, is calculated by backwards propagation and d_{out} by forwards propagation. This analysis can implemented as the two signal interpretation functions `fork` and `gate` as shown in figure 3.22. The signal values have the form of a pair

$$\textbf{type}\ \texttt{Delay} \quad = \quad \texttt{Term}\ (\texttt{Int}, \texttt{Int})$$

where the first element is the fanout and the second element is the estimated delay.

To run a delay analysis of `sklansky` for 8 bits, we define

```
delaySkl d  =  do
  circ       ← sklansky fork gate d
  is_fo      ← sequence (replicate (2^d) free)
  os_del     ← sequence (replicate (2^d) free)
  north circ === edge [just (pair i_fo (int 0))   | i_fo  ← is_fo ]
  south circ === edge [just (pair (int 1) o_del)  | o_del ← os_del]
  return (segs (south circ))
```

Here, we instantiate `sklansky` with the signal interpretations from our analysis, and place the additional constraints that the delay on each input (the north port) is 0 and the fanout on each output is 1.

Running `delaySkl` in GHCi gives the following result:

```
*Main> runLP (delaySkl 3)
[[Just (1,0),Just (1,109),Just (1,212),Just (1,224),Just (1,327)
,Just (1,327),Just (1,327),Just (1,327)]]
```

Since we have only used deterministic operations, we get exactly one result from the computation. The reported numbers have no absolute meaning, but could potentially be used to compare different implementations of a circuit in a way that takes gate fanout into account.

3.5 Discussion

The first embedding of logic programming in Haskell, by Seres and Spivey [SS99], was extended by Claessen [CL00] to use *monads* and *typed* logical variables. In Claessen's approach, a logical data type for lists is introduced with the declaration:

$$\textbf{data}\ \texttt{List}\ a\ =\ \texttt{VarL}\ (\texttt{Var}\ (\texttt{List}\ a))\ |\ \texttt{Nil}\ |\ a :::\ \texttt{List}\ a$$

Then the new type must be instantiated under the `Free` and `Unify` classes (containing a total of 3 methods). Our representation of logical terms has several advantages:

- *Simplicity.* We can introduce a new logical data type very easily, without having to instantiate various type classes.
- *Clarity.* We have a clear correspondence between normal Haskell data types, of type a, and logical data types of type `Term` a. Claessen must invent new, unfamiliar names for logical data types. Our approach facilitates conversion between logical terms and normal Haskell values.
- *Abstraction.* Claessen exposes implementation details of his library, such as the `Var` data type, to the programmer. If the programmer **case**-deconstructs a logical term, how should they handle the `Var` case? Our abstraction prevents this abuse of logical terms.

Claessen does not discuss how to handle primitive Haskell types, such as integers and arithmetic, which we support conveniently using *residuation*. Furthermore, we have shown how *pattern matching* can be achieved, allowing simpler definitions of predicates. Overall, we believe these improvements make the prospect of embedded logic programming much more attractive.

Predicates written using our approach look similar to corresponding Prolog predicates. One difference is that our logical variables are *explicitly quantified* and *typed*. But the main difference is that our approach is *embedded* in Haskell. In Wired, this lets us use logic programming features where they are needed, and to use normal Haskell everywhere else. In particular, our embedding allows logical variables, unification, and residuation together with (higher order) functions, algebraic types, type classes, and existing Haskell libraries such as Lava. Similar features, apart from type classes and Lava, are available in Curry [HKMN95]. Indeed, our first version of MiniWired was developed in Curry.

Moving MiniWired over to our library was straightforward and its run-time performance under each compiler (MCC and GHC) was similar.

3.6 Alternative term representation

The next chapter describes the full implementation of Wired using the LP library. During our work on the implementation, we found that the universal term representation was a bit too heavy-weight to be practical (for example, see the disturbing use of `unpair` in figure 3.22). An alternative representation of terms is

```
data NewTerm a  =  Val a
                |  Var (Ref (Status a))

data Status a   =  Bound (NewTerm a)
                |  Unbound [Residual a]

unterm  ::  Logical b ⇒ NewTerm a → (a → LP b) → LP b

...
```

That is, a `NewTerm` is either a regular value or a logical variable[2]. Deconstruction is done using `unterm`[3]. It corresponds roughly to `rigid` from section 3.3.7, but an important difference is that the continuation now recieves an actual a rather than a `Term` a.

This representation allows a more fine control over the mix between logical terms and normal Haskell values. Imagine a program where the following value occurs:

```
val  ::  ([Int], [Int])
```

The only thing the type tells us is that we have two lists of arbitrary length containing arbitrary integers. Say we want to place additional constraints on the value using `LP` and the `Term` type. Then the only option is to model the value as

```
valTerm  ::  Term ([Int], [Int])
```

which means that the whole value is represented as a universal term, and thus allows arbitrary constraints to be placed on arbitrary sub-terms – for example:

```
constraint  =  unpair valTerm (λas bs → do
                 (x, y)  ←  free
                 as      === int 1 ▷ y ▷ int 3 ▷ x
                 bs      === y ▷ x)
```

[2] Since `Ref` is a typed reference, we cannot use `IntMap` for variables anymore. In our implementation, `Ref` is simply an alias for `IORef`.

[3] `unterm` is quite different from the function with the same name in section 3.3.9, which is only to be used when running the computation.

But what if we just want to say that the lengths of the lists are equal? Then it seems a bit excessive to have logical constructors throughout the whole term, and the price for that is cumbersome deconstruction using functions like `unpair` and `caseOf`. With `NewTerm`, we could instead model the value as

```
valNew :: (NewTerm [Int], NewTerm [Int])
```

This value is very similar to the original `var` – the only difference is that the lists now have the possibility of being variables instead of normal lists. This in turn means that the only place where we can place useful constraints is over the whole lists – for example:

```
constraint = let (as, bs) = valNew
             in unterm as (λas' →
                  unterm bs (λbs' → do
                    guard (length as' == length bs')
                    return (int 1)))
```

This constraint waits until both lists are instantiated, checks that the lengths are equal, and returns `int` 1 if this is the case. Note that since as' and bs' have type `[Int]`, we can use the normal `length` function for lists rather than a special function for terms.

The advantages of `NewTerm` are clear: logical features only where needed, no universal term representation, easier deconstruction, etc. But there are also (minor) disadvantages. If we want to have the ability to place constraints anywhere in the term, we have to insert the `NewTerm` constructor throughout the whole type:[4]

```
valNew' :: NewTerm (NewTerm [NewTerm Int], NewTerm [NewTerm Int])
```

Not only does this look a bit clumsy, it is also "less logical" than `valTerm`. This is because in `valTerm`, every constructor along the list spine is also a logical term, while in `valNew'`, this is just the case for the top-most constructor. To become equivalent to `valTerm`, we would have to have a special list type with the `NewTerm` constructors baked-in

```
data List a = Nil
            | a ::: NewTerm (List a)
```

and then use the following value type:

```
valNew'' :: NewTerm
              (NewTerm (List (NewTerm Int))
              ,NewTerm (List (NewTerm Int))
              )
```

[4] In most cases, we do not need a `NewTerm` on the tuple, because it already has logical features through the `Logical` class (section 3.3.5).

It turns out that for Wired it is usually sufficient to use rather coarse grained term structures; normal tuples and "termed" lists like in `valNew` seems to work most of the time. So going over to the new term representation was a really good move that made the code a lot simpler, both in the implementation and the resulting circuit descriptions. We will see more about this in chapter 4.

```
newtype BS s a = BS { run :: ∀b. s → ((s, a) → b → b) → b → b }

instance Monad (BS s) where
  return a = BS (λs k_s k_f → k_s (s, a) k_f)
  m >>= f  = BS (λs k_s k_f → run m s (λ(s', a) k_f →
                                  run (f a) s' k_s k_f) k_f)

instance MonadPlus (BS s) where
  mzero     = BS (λs k_s k_f → k_f)
  m_0 ⊕ m_1 = BS (λs k_s k_f → run m_0 s k_s (run m_1 s k_s k_f))

get       :: BS s s
get       = BS (λs k_s k_f → k_s (s, s) k_f)
put       :: s → BS s ()
put s     = BS (λ_ k_s k_f → k_s (s, ()) k_f)
```

Figure 3.14: An efficient continuation-based `BS` monad

```
type IntMap a      = [(Int, a)]

empty              :: IntMap a
empty              = []

insert             :: Int → a → IntMap a → IntMap a
insert i a []      = [(i, a)]
insert i a ((j, b) : ps)
  | i == j         = (j, a) : ps
  | otherwise      = (j, b) : insert i a ps

(!)                :: IntMap a → Int → a
((i, a) : ps) ! j  = if i == j then a else ps ! j
```

Figure 3.15: Specification of the `IntMap` data structure

```
instance Conv Bool where
  unterm (Term (Ctr 0 [])) = return False
  unterm (Term (Ctr 1 [])) = return True
  unterm _                 = mzero

instance Conv Int where
  unterm (Term (Int a))    = return a
  unterm _                 = mzero

instance Conv a ⇒ Conv (Maybe a) where
  unterm (Term (Ctr 0 [])) = return Nothing
  unterm (Term (Ctr 1 [a])) = liftM Just (unterm (Term a))
  unterm _                 = mzero

instance Conv a ⇒ Conv [a] where
  unterm (Term (Ctr 0 [])) = return []
  unterm (Term (Ctr 1 [a, b])) =
    liftM2 (:) (unterm (Term a)) (unterm (Term b))
  unterm _                 = mzero

instance (Conv a, Conv b) ⇒ Conv (a, b) where
  unterm (Term (Ctr 0 [a, b])) =
    liftM2 (,) (unterm (Term a)) (unterm (Term b))
  unterm _                 = mzero
```

Figure 3.16: Instances of the Conv class

```
instance Logical (Edge a) where
  free        = do n ← free ; as ← free
                   return (Edge n as)
  a === b     = do len a  === len b
                   segs a === segs b
  match a b k = match (len a) (len b)
                  (match (segs a) (segs b) k)

instance Logical (Tile a) where
  free        = do (n, s) ← free
                   (e, w) ← free
                   (x, y) ← free
                   return (Tile (Edge y w) (Edge x n)
                                (Edge x s) (Edge y e))
  a === b     = north a === north b ≫ east a === east b
              ≫ south a === south b ≫ west a === west b
  match a b k = match (north a) (north b)
                  (match (east a) (east b)
                    (match (south a) (south b)
                      (match (west a) (west b) k)))
```

Figure 3.17: The `Edge` and `Tile` instances of the `Logical` class

```
wireT3 f    = do a       ← free
                 (a0, a1) ← f a
                 let w    = edge [nothing]
                 let n    = edge [just a]
                 let s    = edge [just a0]
                 let e    = edge [just a1]
                 return (Tile w n s e)

crodot f g  = do a       ← free
                 b       ← free
                 (a0, a1) ← f a
                 c       ← g a0 b
                 let w    = edge [just a]
                 let n    = edge [just b]
                 let s    = edge [just c]
                 let e    = edge [just a1]
                 return (Tile w n s e)

wireY1      = do a       ← free
                 let ns   = edge [just a]
                 let we   = edge [nothing]
                 return (Tile we ns ns we)

thinX1      = do a       ← free
                 let ns   = edge [just a]
                 let we   = edge []
                 return (Tile we ns ns we)

thinY       = do we      ← free
                 let ns   = edge []
                 return (Tile we ns ns we)
```

Figure 3.18: Remaining definitions of the primitive tile set

```
(*=*) :: Desc a → Desc a → Desc a
d0 *=* d1 = do t0 ← d0 ; t1 ← d1
               e ← join (east t0) (east t1)
               w ← join (west t0) (west t1)
               north t0 === south t1
               return (Tile w (north t1) (south t0) e)
```

Figure 3.19: The "below" combinator

```
a ⁻ b  =  unint a (λa' → unint b (λb' → sub a' b'))
  where
    sub a b  =  if a>b then mzero
                       else return (int (a−b))
```

Figure 3.20: Non-negative subtraction

```
row    ::  Desc a → Desc a
row d  =   do n  ←  free
              s  ←  unint n (λn' → case n' of
                                     0  → thinY
                                     _  → d *||* row d)
              n === len (north s)
              return s
```

Figure 3.21: Fully-deterministic definition of `row`

```
type Delay  =  Term (Int, Int)

unpair p f  =  do (a, b) ← free
                  p === pair a b
                  f a b

calc d0 d1 n  =
    unint d0 (λd0 →
      unint d1 (λd1 →
        unint n (λn →
          return (int (max d0 d1+100+3∗n)))))

fork     ::  Delay → LP (Delay, Delay)
fork a   =   do (bfo, bdel) ← free
                (cfo, cdel) ← free
                unpair a (λafo adel →
                  do afo <== bfo ∔ cfo
                     bdel === adel
                     cdel === adel
                     return (pair bfo bdel, pair cfo cdel))

gate     ::  Delay → Delay → LP Delay
gate a b =   do unpair a (λafo adel →
                  unpair b (λbfo bdel →
                    do afo === int 1
                       bfo === int 1
                       cfo ← free
                       cdel ← calc adel bdel cfo
                       return (pair cfo cdel)))
```

Figure 3.22: A simple bi-directional delay analysis by non-standard interpretation

Chapter 4

Reimplementing Hardwired

In this chapter, we look at a reimplementation of Hardwired (from chapter 2) using the logic programming library developed in chapter 3. An important difference in this implementation is the separation of the structural and geometrical circuit aspects. Section 4.1 presents the structural view, and section 4.2 adds geometry on top of this. Note that this implementation is not used anymore; the current Wired implementation is presented in part II of the thesis.

4.1 A relational Lava

In the introduction, we discussed the usefulness of a relational circuit model, allowing circuit descriptions to be written without regard to the direction in which signals are flowing. Hardwired is based on such a model, but the initial implementation was not very satisfactory (section 2.4). One problem was the fact that the structural and geometrical view of the circuit were tightly intertwined, making things unnecessarily complicated. In a more modular implementation of Hardwired, we would like these aspects to be implemented separately.

The structural circuit view is going to be provided by a Lava-like library supporting logical features from our logic programming library (LP). The basic circuit type is `Signal`, roughly corresponding to the type with the same name in Lava. `Signal`s can be constructed using a set of predefined constants and gates[1], such as

```
low, high :: Signal
inv       :: Signal → LP Signal
and2, or2 :: (Signal, Signal) → LP Signal
```

[1]The set of gates is extensible and can be arranged into libraries.

For example, a 1-bit multiplexer can be defined as

```
mux :: (Signal, (Signal,Signal)) → LP Signal
mux (sel, (a,b)) = do
    sel' ← inv sel
    x1   ← and2 (sel',a)
    x2   ← and2 (sel, b)
    or2 (x1,x2)
```

The difference between this version and the following one in Lava

```
mux (sel, (a,b)) = or2 (and2 (inv sel, a), and2 (sel, b))
```

is purely syntactical. The former uses a monadic style (explained in section 3.2.1), and that is the price we pay for having signals with logical features.

The logical features are provided through the *Logical* class from section 3.3.5, which means that *Signal*s support the following operations:

```
free  :: LP Signal
(===) :: Signal → Signal    → LP ()
(<==) :: Signal → LP Signal → LP ()
```

4.1.1 Signal flow abstraction

As a demonstration of signal flow abstraction provided by the *Logical* interface, here is a redefinition of `mux` with the components instantiated in reversed order:[2]

```
muxRev (sel, (a,b)) = do
    (sel',x1,x2) ← free
    x      ← or2 (x1,x2)
    x2   <== and2 (sel, b)
    x1   <== and2 (sel',a)
    sel' <== inv sel
    return x
```

From section 3.2.1, we remember that the lines in a do-block are syntactically expanded using the `>>=` operator. This means that in `mux`, signals are flowing forwards through `>>=`, while in `muxRev`, they are flowing backwards.

For a different example, let us try to take the "converse" of a circuit:

```
converse circ output = do
    input    ← free
    output <== circ input
    return input
```

This combinator takes the circuit's output as argument and returns its input. Such a combinator has also been explored by Claessen [Cla00], but in his ap-

[2] The notation can be slightly misleading in examples like this. Note that `x ← m` is a *binding construct* which binds `x` to the result of `m`, while `x <== m` is just an application of the operator `<==` to the arguments `x` and `m`.

proach, relational definitions would generate constraints for an external solver rather than being handled internally, as in relational Lava. Using `converse`, we can now define the following two equivalent circuits:

```
serialInv     = inv >=> inv
serialConvInv = converse (converse inv >=> converse inv)
```

Here, the serial composition operator `>=>`[3] has abstract signal flow; in the first case the signal flows to the right, in the second case it flows to the left.

It is easy to come up with such contrived examples of abstract signal flow, but cases where this gives a definite advantage are hard to find. For a concrete example, that could arise in a real situation (see section 9.5), consider the following flat representation of placed netlists:

```
type Position = (Int,Int)

data Comp = Comp
       { gate :: [Signal] → LP Signal
       , inps :: [Position]
       , out  :: Position
       }

data Circuit = Circ
       { inputs  :: [Position]
       , outputs :: [Position]
       , comps   :: [Comp]
       }
```

A component (*Comp*) is defined by its logical function and the positions of its inputs and outputs. A *Circuit* is a list of such components, with additional information about the positions of the primary inputs and outputs. One possible representation of our `mux` in this format is given in figure 4.1, and the corresponding layout in figure 4.2. Each signal corresponds to a unique position, and gates are placed at the same position as their output signal.

Now we want to define a function for constructing a relational Lava circuit from such a description:

```
buildCirc :: Circuit → ([Signal] → LP [Signal])
```

An imperative solution is to keep a table remembering the computed *Signal* at each position and use the following scheme:

1. Insert all inputs into the table.
2. For each component, lookup its inputs, apply the logical function, and insert its outputs.
3. Lookup all outputs.

[3] This is a standard Haskell combinator. For circuits, it works just like `->-` in Lava.

```
inv' [a]   = inv a
and' [a,b] = and2 (a,b)
or'  [a,b] = or2 (a,b)

muxCirc = Circ { inputs  = [sel,a,b]
               , outputs = [x]
               , comps   = [ Comp inv' [sel]    sel'
                           , Comp and' [sel',a] x1
                           , Comp and' [sel,b]  x2
                           , Comp or'  [x1,x2]  x
                           ]
               }
  where
    sel  = (0,0); a  = (1,0); b  = (2,0)
    sel' = (0,1); x1 = (1,2); x2 = (2,1)
    x    = (2,3)
```

Figure 4.1: Definition of multiplexer in the *Circuit* format

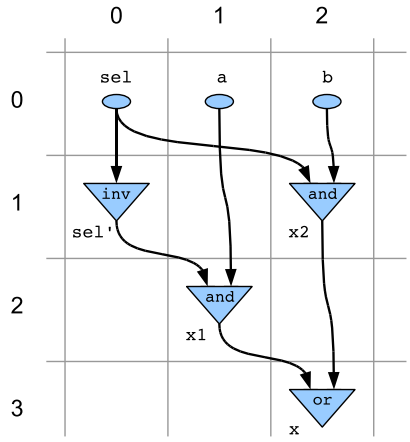


Figure 4.2: Layout of `muxCirc`

However, this method has the obvious flaw that it assumes the inputs of each component to be inserted before the component is converted, so we would have to run a topological sort before the conversion in order to guarantee success.

Another, more declarative solution is to make use of the fact that *Signal*s can be logically quantified. In fact, we can create the whole table in one go using the following function:

```
declareSigs :: Circuit → LP (Map Position Signal)
declareSigs circ = liftM fromList (mapM declare sigs)
  where
    sigs        = concat [o:is | Comp _ is o ← comps circ]
    declare sig = liftM2 (,) (return sig) free
```

This uses the standard data structure *Data.Map* [Hoo] to represent the table. Here, `sigs` is the list of all positions in the circuit, `mapM declare` associates each signal with a free variable, and `liftM fromList` creates a map from the list of associations. This function can now be used to define a robust `buildCirc`:

```
buildCirc circ ss = do
    posMap  ← declareSigs circ
    let pos = (posMap !)
    sequence_ [ pos i === s | (i,s) ← zip (inputs circ) ss ]
    mapM_ (buildComp pos) (comps circ)
    return (map pos (outputs circ))
  where
    buildComp pos (Comp gt is o) = pos o <== gt (map pos is)
```

Here, `pos` is a lookup function for the table obtained from `declareSigs`. The three last lines in the do-block correspond to the three steps in the above scheme, except that the information flow in step 2 is now abstract. In fact, since LP computations are commutative with respect to monadic sequencing, the three steps need not even be done in order in this version. Moreover, a topological sort would not work with feedback loops, while in this version, these are handled just fine.

We will come back to this example in section 7.1.2.

4.1.2 Implementation

Lava 2000 uses a tree representation of circuits, where each node corresponds to a gate, and the leafs correspond to constants or primary inputs [Cla00]. A node in this tree is represented by the following data type:

```
data S s = Bool Bool
         | VarBool String
         | Inv s
         | And [s]
         | ...
```

The constructor *Bool* makes a Boolean constant, *VarBool* makes a Boolean input symbol; `inv` and `and` are gate nodes whose inputs are the given arguments. This type is wrapped up in the following way:

```
newtype Symbol = Symbol (Ref (S Symbol))
```

This defines a recursive tree type by forcing the gate inputs to be of type *Symbol*. The additional *Ref* constructor allows nodes to be uniquely identified in order to detect sharing. This means that *Symbol* can be interpreted as a directed graph. Finally, the *Signal* type is simply a *Symbol* with a phantom type argument (section 3.3.4):

```
newtype Signal α = Signal Symbol
```

Our relational Lava has a similar data type, except that we use *Term*s (the alternative representation from section 3.6) and omit the phantom type:[4]

```
data Signal = Signal (Term Int) (Term (S Signal))
```

The *Term Int* has the same role as *Ref* in Lava, that is, we treat two signals with the same unique number as identical. Since the type is essentially a pair of *Term*s, it can be instantiated under the *Logical* class in the same way as normal pairs (section 3.3.5).

It is important to note that, altough the library is called relational Lava, the relations only exist at the *generator level*. The circuits that are generated have a functional signal flow, which can be seen from the structure of the *S* type.

Instead of implementing simulation for relational Lava, we chose to make a conversion function that converts from relational to normal Lava. The advantage of this is that we can reuse all of Lava's backends, including links to verification engines and VHDL generation. However, the timing analysis from section 2.2.2 was implemented directly in relational Lava. Format conversion and timing analysis turned out to be expressible in terms of a common framework. This technique has now been refined and is presented in chapter 7.2.4, which is why we leave out the details here.

One problem with this implementation is that it doesn't work directly for circuits with feedback loops. An example of such a circuit is

```
toggle = do
    x  ← free
    x' ← inv x
    x <== delay x'
    return x
```

The reason why it doesn't work is that `deref` (section 3.3.9) goes into an infinite loop when recursing through the tree. Even though we interpret *Signal*s as directed graphs, they are still just trees to functions like `deref`. We have tried

[4] This definition is inspired by a simple proof-of-concept implementation of relational Lava by Matthew Naylor.

a couple of different solutions to this. One was to cut all loops before calling `runLP` and then reintroduce them afterwards. This worked well, but felt like an unsatisfactory hack. Another solution was to make `deref` lazier. This also worked, but it used an unsafe hack which could potentially lead to unexpected program behavior.

An alternative to relational Lava, which is simpler and also overcomes the problem with feedback loops, is presented in chapter 7.

4.2 Hardwired

4.2.1 Implementation

Hardwired is implemented as a geometrical layer on top of relational Lava. The main ideas have already been given in the presentation of MiniWired in section 3.4, so here we will just summarize the main additions and improvements:

- The behavior/structure is represented using relational Lava instead of using separate parameters as in figure 3.18.
- Descriptions contain a representation of their layout. This representation can be converted to a postscript picture, such as the ones in figure 4.3. In principle, it could also be exported to formats for circuit layout (such as CIF or GDSII), but this has not been implemented because these formats require more detail than what seems suitable for Hardwired (see the discussion in chapter 5).
- The "edges" of a circuit can be arbitrary structures, not just lists.
- The type of the edges are reflected in the circuit type. For example,

  ```
  Circ (Sig,Sig) Ins [(Ins,Sig)] Ins
  ```

  is the type of a circuit whose western port is a pair of signals, north port is a single insulator, and so on[5]. For another example, the "beside" combinator,

  ```
  (*||*) :: (...)
      ⇒ Circ wL nL sL x
      → Circ x nR sR eR
      → Circ wL (nL,nR) (sL,sR) eR
  ```

  accepts any two circuits for which the east port of the first one has the same type as the west port of the second one. The returned circuit has the original north ports grouped as a pair, and similarly for the south ports.

[5] Comparing to section 2.1.1, *Ins* corresponds to an i-contact and *Sig* to an s-contact.

- Wire blocks are *generic*. This means that they expand in the "obvious" way when applied in a context of compound signals. The two layouts in figure 4.3 have the same generic wires – only the operators are different.
- Geometry is modeled in more detail. Instead of using a coarse grid based on the component size, sizes now accurately reflect the actual dimensions of cells and wires.

4.2.2 Sklansky

A direct implementation of Sklansky (from sections 2.3.2 and 3.4.2) in the new Hardwired looks as follows:

```
sklansky 0 op opF0 = rowN 1 $
    nilX `satisfyingN` (<== (structure =<< asLP south =<< op))

sklansky d op opF0 = (joinL ~||~ joinR) *=~ (skl~||~skl)
  where
    skl    = sklansky (d-1) op opF0
    n      = term (2^(d-1) - 1)
    op'    = op   *=* alignLeft 200
    opF0'  = opF0 *=* alignLeft 200
    joinL  = rowN n wire   ~||* (wireT1*=*wire)
    joinR  = rowN n opF0'  ~||* op'
```

The extra complexity compared to the previous versions comes from the fact that this description is more general and takes more details into account. For example, the wire blocks (`alignLeft`, `wire` and `wireT1`) are now generic and work for both single-bit operators, as in figure 4.3(a), and two-bit oparators[6], as in figure 4.3(b).

4.2.3 Multiplier reduction trees

Binary multiplication can be divided into two steps: generating partial products, and adding the partial products. For example, to perform the multiplication $1011_2 \cdot 0101_2$ $(11 \cdot 5)$, we first generate the partial products

```
  1011
 0000
1011
0000
```

and then add them up to get the number 00110111_2 (55). Adding the partial products can be done in different ways. One way is to count all ones in each column above from right to left. Each column will then get one result bit and a number of carries to the next column. The bits in each column can be added

[6] The component used for computing addition carries operates on pairs [BK82].

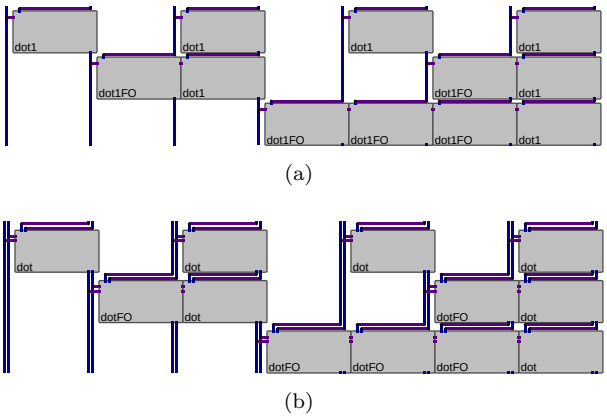


Figure 4.3: (a) Sklansky with single-bit operator (b) Sklansky with pair operator

linearly or in a tree-like fashion. Networks of the former kind are commonly called *reduction arrays*, and networks of the latter kind are called *reduction trees*. We will also use *array* to denote the union of these two kinds.

Eriksson et al. have developed a multiplier reduction tree with logarithmic depth and regular connectivity [Eri03, ELES+06] which is very suitable for description in a language like Wired. An interesting property of the underlying method is that it can produce a variety of different reduction arrays by just altering small parts of the wiring. Sheeran gives a simple implementation of this method in Lava (including some extensions) [She04], and it turns out that her recurrence equations can be transferred to Hardwired quite easily.

Figure 4.4 shows the general structure of a reduction array for adding the partial products of a 6-bit multiplication in the above method. This picture has the least significant bit on the left. The `H` blocks are half-adders and the `F` blocks are full-adders. Each of these blocks processes three (`F`) or two (`H`) incoming bits (including carries), and the remaining bits are just passed down to the next level. The task of the `W` block is to insert the carry from the left in the list of bits from the top. The exact wiring in the blocks may be chosen differently in order to achieve different array structures. We see that each column reduces its incoming partial product bits and carries until there are only two bits left. The reason for stopping at two bits is that we can get a more efficient multiplier by using a dedicated adder for these last bits. We will now see how this structure, with detailed routing, can be described in Hardwired.

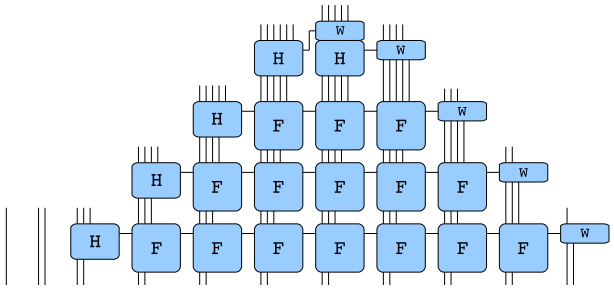


Figure 4.4: Structure of reduction array for 6-bit multiplication

Top-level definition

To allow a modular description that separates the main recursion from all the geometrical details, we use an intermediate representation of the reduction tree as a martix of blocks `[[Block]]`, where *Block* is defined as

```
data BlockType = W | WH | H | F | Nil
type Block     = (BlockType, Int)
```

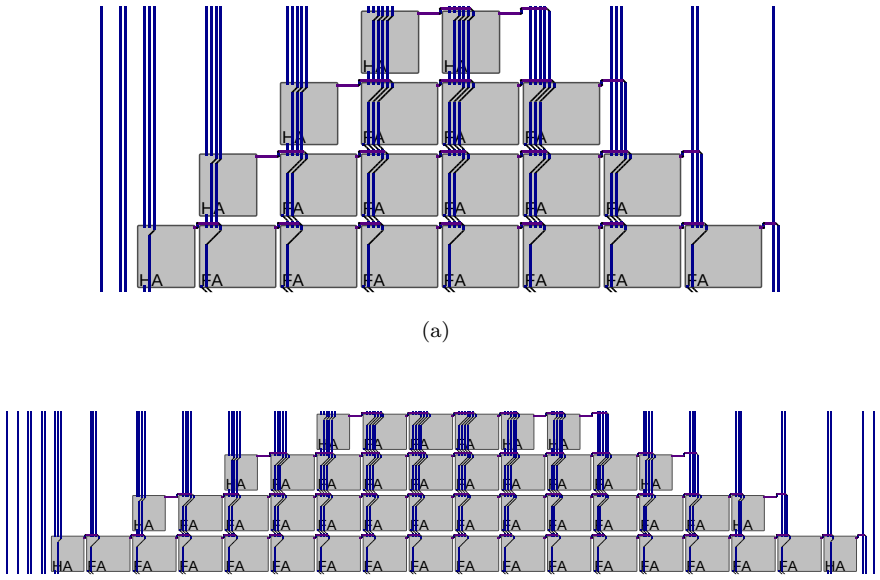
where the different block types correspond to the ones in figure 4.4. *WH* is a merged *W* and *H* block. Sheeran's algorithm can quite easily be modified to obtain the function

```
redArrayBlock :: [Int] → Int → [[Block]]
```

The first argument is the number of bits at the top of each column, and the second number is the number of carries from the left. Here is an example evaluation:

```
*Main> redArrayBlock [1,2,3,4,5,6,5,4,3,2,1] 0
[ [(Nil,1)]
, [(Nil,2)]
, [(H,3)]
, [(H,4),(F,3)]
, [(H,5),(F,4),(F,3)]
, [(H,6),(F,5),(F,4),(F,3)]
, [(WH,5),(F,5),(F,4),(F,3)]
, [(W,4),(F,5),(F,4),(F,3)]
, [(W,3),(F,4),(F,3)]
, [(W,2),(F,3)]
, [(W,1)]
]
```

This represents the array in figure 4.4. Each column is listed from top to bottom. The resulting reduction tree, after translation to Hardwired, is shown in figure 4.5(a). To demonstrate the generality of the method, we also show a tree



(a)

(b)

Figure 4.5: Multiplier reduction trees

with the following list of partial product bits (which results from using simple Booth encoding in a 12-bit multiplication):

```
[1,1,2,2,3,3,4,4,5,5,6,6,6,6,5,5,4,4,3,3,2,2,1,1]
```

The resulting reduction tree is shown in figure 4.5(b).

The code that translates the matrix representation to Hardwired is given in figure 4.6. We will not go into the details here, just make a few remarks. First of all we see the use of complex logical terms with the new term representation from section 3.6. For example, the *`Side`* type says that the side of a block is either an *`Ins`* or a pair *`(Ins,Sig)`*. By using an explicit *`Term`* constructor on top of this, we turn this regular Haskell type into a logical term. The function `redArray` builds the whole array out of the matrix returned from `redArrayBlock`. It uses `listRow`, which is a version of `row` that composes a list of components rather than a repetition of a single component. The helper function `compress` builds one column in the tree using `listCol`, which is the vertical correspondence to `listRow`. Finally, `mkBlock` builds a single block in a column.

```
type Side = Term (Either Ins (Ins,Sig))
type ISSI = (Ins, Term [Sig], Ins)

left'  = left part0
right' = right part0

mkBlock :: Size → Block → Circ Side ISSI ISSI Side
mkBlock width (W, x) = convW right' $ convE left'  $ wBlock2  width x
mkBlock width (WH,x) = convW right' $ convE right' $ whBlock2 width x
mkBlock width (H, x) = convW left'  $ convE right' $ hBlock2  width x
mkBlock width (F, x) = convW right' $ convE right' $ fBlock2  width x

compress :: Int → [Block] → Circ (Term [Side]) ISSI ISSI (Term [Side])
compress yTot blocks = case head blocks of
    (Nil,_) → wire' (wireLen+1)
    _       → compr (reverse blocks) ~=~ wire' wireLen
  where
    wireLen = yTot - length blocks
    x       = snd $ head blocks
    width   = maximum $ map blockWidth blocks

    wire' len = wire `satisfying` λcct → do
        term len <== rLengthR (west cct)
        mapR (<== (liftM (term . Left) $ ins blockHeight)) (west cct)
        south cct <== (liftM north $ wBlock width x)
        return ()

    compr = listCol . map (mkBlock width)

redArray :: [Int] →
    Circ (Term [Side]) (Term [ISSI]) (Term [ISSI]) (Term [Side])
redArray xs = listRow $ map (compress yTot) blocks
  where
    blocks = redArrayBlock xs 0
    yTot   = maximum $ map length blocks
```

Figure 4.6: Definition of `redArray` which builds a Hardwired circuit from the matrix representation of reduction trees

Detailed block definition

So far, the code is quite manageable, but the most difficult part still remains: the definition of the individual blocks, `wBlock2`, `whBlock2`, `hBlock2` and `fBlock2`. This is where things start to get really complicated. Figure 4.7 shows the definition of `hBlockInner`, which is a part of the definition of `hBlock2`. It is quite clear that this code is much more complicated than we would like for such a simple component. A closer look shows that most of the code is actually concerned with the information that flows through the edges of blocks. For example, we see several definitions like

```
convN cn $ convS cs $ circ
```

This applies the conversion relation `cn` to the north port and `cs` to the south port of `circ`. Interestingly, if we remove everything that has to do with port conversion, the whole definition simplifies to

```
hBlockInner w nTop = wiring `over` cell
  where
    cell = halfAdd *||* wire

    wiring = wire *=* wir2 *=* wire
      where
        wir2 = space *||* shiftLeft *||* space
```

This is not type-correct, and lacks the necessary size constraints, but at least the layout aspect is quite clear: the block consists of some wiring on top of a cell. The cell is a half-adder with a wire on its right (the horizontal carry signal coming out of each half-adder in figure 4.5). The wiring consists of three parts placed above each other: first some straight wires, then a small diagonal segment, and finally a small segment of straight wires.

So why do we need all the tedious conversion in `hBlockInner`? This can be most easily understood by looking at a similar situation. Let us go back to the definition of `adder` in section 1.3:

```
adder (cIn, [])        = ([],   cIn)
adder (cIn, (a,b):abs) = (s:ss, cOut)
  where
    (s,c)     = fullAdd (cIn, (a,b))
    (ss,cOut) = adder (c,abs)
```

This definition is also mostly concerned with threading the right data to the right place; the pattern matchings (which take data apart) can be seen as one kind of conversion, and the tuple and list constructions (which put data back together) are another kind of conversion. However, the data threading in `adder` is very clear and readable compared to the complex expressions in `hBlockInner`. There are several reasons for this:

- Hardwired ports are logical terms, which means that manipulation has some overhead. In general, we also want the information flow through

```
hBlockInner :: Size → Int →
    Circ Ins (Ins, Term [Sig], Ins) (Ins, Term [Sig], Ins) (Ins,Sig)
hBlockInner w nTop = convN cn $ convS cs $ wiring `over` cell
  where
    cn = triple (part1, part2 </ part2' </ part2, part2)
    cs = triple (part1, part1 </ part2, part2)

    cell :: Circ Ins (Ins, (Sig,Sig), Ins) (Ins, Sig, Ins) (Ins, Sig)
    cell = convN (c 3) $ convS (c 2) $
        halfAdd *||* (wire :: Circ (Ins, Sig) Ins Ins (Ins, Sig))
      where
        c n = triple
          (part2, part2, toIns part0 `ofSize` term (w - n*sigSize))

    wiring ::
      Circ Ins (Ins, Ins, (Term [Sig], Ins))
           (Ins, Ins, (Term [Sig], Ins)) Ins
    wiring = insW $ insE $ wir1 *=* wir2 *=*
             (wir1 `satisfyingW` (<== ins sigSize))
      where
        wir1 ::
          Circ Ins (Ins, Ins, (Term [Sig], Ins))
               (Ins, Ins, (Term [Sig], Ins)) Ins
        wir1 = wire

        wir2 ::
          Circ Ins (Ins, Ins, (Term [Sig], Ins))
               (Ins, Ins, (Term [Sig], Ins)) Ins
        wir2 = convN cn $ convS cs $ space *||* shiftLeft *||* space
          where
            cn = triple
              ( cutIns sigSize part2
              , ((part2 `ofSize` sigSize)-+-(part3 `ofSize` sigSize))
              , pair (part3'',part1)
              )

            cs = triple
              ( cutIns sigSize part2
              , part2 `ofSize` sigSize
              , pair (part3, toIns part0)
              )
```

Figure 4.7: Part of the definition of `hBlock2`

the conversions to be bidirectional. To ensure this property (and some other ones), all conversions are defined using primitives from a conversion library. For example, `cn` and `cs` in figure 4.7 (there are two definitions of each) are defined using such primitives.

- The ports contain more information; we also need to propagate structural information for sizes to be inferred correctly.
- Hardwired circuits are built using combinators that connect the components according to a predefined pattern. For example, the `*||*` combinator always connects the east/west ports and always returns a circuit with pairs on the north and south ports. In contrast, `adder` uses named signals to connect its sub-components in the appropriate way.

To sum up the example: We have translated a Lava description of a reduction tree to Hardwired. From a top-level perspective, the description is quite nicely recursive and highly parameterized just like the Lava version. However, the building blocks are much more complicated. The amount of detail needed in these definitions makes them very hard to get right and almost impossible to change without doing a complete rewrite. The conclusion has to be that Hardwired is not suitable for descriptions with this amount of detail. A more useful version of Wired is therefore developed in the second part of the thesis.

Chapter 5

Discussion – part I

In this part of the thesis, we have seen the early work on Wired (Hardwired) with some examples of what we want to achieve with the system. The system gives some promise of allowing concise descriptions of regular layouts including wiring details. Timing analysis is built in, and there seems to be hope of using timing results to guide the generation of circuits – similar to previous work in Lava, but with more accuracy.

We also saw some major improvements to the implementation enabled by the LP library. This had a great impact on the efficiency and user-friendliness of the system. In fact, given the detailed circuit model used in Hardwired, there doesn't appear to be very much room for improvement on the implementation side. However, the original idea behind Wired was to have a system which allows elegant Lava-like descriptions with some more awareness of the underlying physical effects. Somehow, Hardwired seems to be quite far from this goal. The multiplier reduction tree is the most advanced example we have tried out in Hardwired. It has roughly the same level of detail as the description in [ELES+06], so it can be said to be quite realistic. However, from the very tedious and complicated code that this requires, we draw the hardheaded conclusion that Hardwired's circuit model is not suitable for description of realistic circuits.

The main problem with the model is the fact that wires are represented explicitly and in great detail. This places too much responsibility on the designer and means that the important aspects of the circuit get buried in the details of the routing. It also makes the descriptions very inflexible, since, in general, small changes to a circuit's structure can cause big changes to the routing. Finally, detailed routing just doesn't seem suited for description by layout combinators in any but the simplest cases.

The other problem is that the model is not completely realistic. For example, while the designer gets detailed control over the routing, there is no guarantee that the specified wire geometries are actually valid. In practice, there are

many so-called design rules to consider (needed to guarantee a manufacturable design), and these are far beyond what we want to deal with in Wired. Another deviation from reality is the way that all communication happens through the edges of adjacent blocks. This means that the routing and the cell layout need to be partitioned in exactly the same way.

To summarize the problems with the model, we can say that it is *too detailed*, and at the same time it captures the *wrong details*. Not surprisingly, the solution appears to be to abandon the explicit representation of wires, and instead represent each wire by a set of points. This approach is explored in the second part of the thesis.

Part II

Wired

Chapter 6

Introduction – part II

In the discussion of the first part of the thesis (chapter 5), we concluded that the main problem with the Hardwired model was the fact that wires were *explicit*, which originated from the goal of a design system with better control over wire effects. However, by sacrificing some of that accuracy and instead viewing each net as a set of connected points, we can get a much more useful system. This was originally intended just as a simpler version of Hardwired, but it actually turned into a very different system with a much improved description style. The most important differences are that wires are now implicit and that the data flow has been decoupled from the layout.[1] Another difference is that the new Wired is not dependent on the LP library (chapter 3). Some features of logic programming are still needed, but this is now achieved using a much simpler library.

This part of the thesis presents this new version of Wired, and shows its implementation and some results of our experiments in the system. But before diving into the implementation, we will go through an example to demonstrate how it will be used.

6.1 Sklansky: Functional version

We are already familiar with the Sklansky network from sections 2.3.2 and 4.2.2. This time we will start from a very simple Lava description:

[1] In chapter 4, we had separate implementations of structure and geometry, but now they are separated in the front-end too.

```
sklansky :: ((α,α) → α) → [α] → [α]
sklansky op [a] = [a]
sklansky op as  = ls' ++ [op (last ls', r) | r ← rs']
  where
    k       = length as `div` 2
    (ls,rs) = splitAt k as
    ls'     = sklansky op ls
    rs'     = sklansky op rs
```

Compared to the previous versions – purely based on layout combinators – this version instead focuses on the data flow. The results from the two recursive calls, `ls'` and `rs'`, are normal Haskell lists. The result is obtained by combining these two lists using `++`, the operator `op` and list comprehension. In fact, this is a normal program in standard functional programming style, and as the type suggests, we can actually use any function `(a,a) → a` as the operator – for example:

```
*Main> sklansky (uncurry (+)) [1..10]
[1,3,6,10,15,21,28,36,45,55]
```

A Lava simulation may look like this:

```
*Main> simulate (sklansky or2) [low,low,low,high,low,high,low]
[low,low,low,high,high,high,high]
```

We think of this as the ideal model of the network – this is the kind of code we would like to write. The Wired implementation uses a monad (section 3.2.1) to keep track of the placement of circuits, so the first step in porting Sklansky to Wired is to turn the Lava code into monadic style:

```
sklansky :: Monad m ⇒ ((α,α) → m α) → ([α] → m [α])
sklansky op [a] = return [a]
sklansky op as  = do
    let k       = length as `div` 2
        (ls,rs) = splitAt k as
    ls'  ← sklansky op ls
    rs'  ← sklansky op rs
    rs'' ← sequence [op (last ls', r) | r ← rs']
    return (ls' ++ rs'')
```

This code is only slightly different from the original version, and so far it is generic – it works for any monad `m`. Running it with the *`Identity`* monad [Hoo] can be done as follows:

```
*Main> runIdentity $ sklansky (return . uncurry (+)) [1..10]
[1,3,6,10,15,21,28,36,45,55]
```

More interestingly, it is in fact a completely valid description in the new Wired system! It can be simulated,

```
*Main> simulate (stripLayout . sklansky and2HSP) [low,low,high,low]
[low,low,high,high]
```

and we could even export a postscript picture of its layout (which would only display a big rectangle since no placement information has been given).

6.2 Refinement: Adding layout annotations

The next step is to refine this description by supplying placement information. Any Wired description can be "annotated" by the layout directives `rightwards`, `leftwards`, `upwards`, `downwards` and `merged`[2], which specify the relative placement of blocks underneath (down to the next annotation). `merged` placement means that the blocks are overlaid on the same area.

The Sklansky code has three top-level "blocks" (the lines containing a ←), and we want the first two (the recursive calls) to be placed beside each other and the last one to be below all the rest (see figure 2.10). Since we want different placements within the code, we first have to group the blocks accordingly:

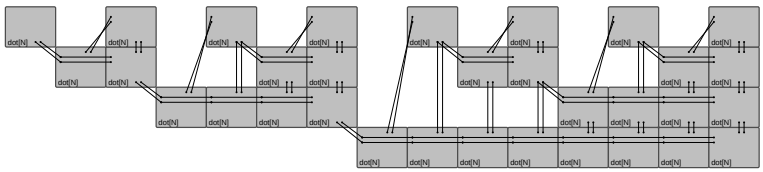
```
sklansky op [a] = return [a]
sklansky op as  = do
    let k       = length as `div` 2
        (ls,rs) = splitAt k as
    (ls',rs')   ← liftM2 (,) (sklansky op ls)
                             (sklansky op rs)
    rs''        ← sequence [op (last ls', r) | r ← rs']
    return (ls' ++ rs'')
```

Here we have grouped the two recursive calls into a single "action" returning a pair of results. Now there are only two blocks on the top level, and these should have a `downwards` placement. The blocks are themselves compound and should have `rightwards` placement internally:

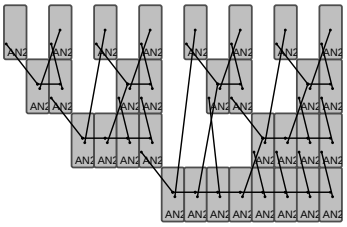
```
sklansky op [a] = space cellWidth [a]
sklansky op as  = downwards 1 $ do
    let k       = length as `div` 2
    let (ls,rs) = splitAt k as
    (ls',rs')   ← rightwards 0 $ liftM2 (,) (sklansky op ls)
                                            (sklansky op rs)
    rs'' ← rightwards 0 $ sequence [op (last ls', r) | r ← rs']
    return (ls' ++ rs'')
```

The numbers given to `downwards` and `rightwards` define the alignment of blocks with unmatching width/height; 0 means towards the *bottom-left* corner and 1 means towards *top-right*. The only place where alignment matters in `sklansky` is on the top level where the first block is wider than the second one. Chosing an alignment of `1` here pushes the second block (the row of `ops`) to the right. We also changed the base-case so that we get horizontal space between the operators on the highest row. This code defines a complete placement for Sklansky, and the result for 16 bits is shown in figure 6.1.

[2]These "directives" are implemented as ordinary Haskell functions.



(a)



(b)

Figure 6.1: Sklansky with two different operators

The picture in figure 6.1(a) was rendered by giving the following command at the GHCi prompt:

```
*Main> renderCircuitNets "skl" $ sklansky dot (replicate 16 low)
```

This produces the file `skl.ps`.

Figure 6.1(a) uses a made-up cell where the pin locations are chosen to make the pictures look clean. For purposes of demonstration, figure 6.1(b) also shows the network with an operator from a real cell library. We see that this network has many more crossing wires. In Hardwired, the extra complexity of routing these crossings would have caused big changes to the `sklansky` code, but in Wired, no changes are needed.

This example demonstrates several features of the new Wired compared to Hardwired:

- Communication between blocks is not bound to occur through the edges; in `sklansky`, signals are communicated via the variables (`ls'`, `rs'`, etc.). In effect, we have separated the data flow from the placement, and this allowed us to define `sklansky` through simple refinement steps.
- Descriptions are much more flexible to changes (see figure 6.1).
- Wires are *implicit*. In figure 6.1(a), the only thing we know about the wires is the locations of the end points of each net, hence all wires are drawn as straight lines. In chapter 9, we will see how the routing can be further constrained by adding "guiding points".

Another important difference, which we will look at in section 9.3 is the fact that Wired descriptions can be exported to standard CAD tools. This is the ultimate proof that the model used is actually realistic.

6.3 More monads

In the rest of the thesis, we will make heavy use of a few different standard monads. Readers familiar with the *Reader*/*Writer*/*State* monads and monad transformers may wish to skip this section. Monads in general are introduced in [Wad93] and section 3.2.

6.3.1 State

In section 3.2, we introduced monads, and in particular, we saw how monads can be used to thread a hidden state through a computation. In Haskell's standard libraries, the *Control.Monad.State* module provides a state monad,

```
newtype State s α = State (s → (α,s))
```

similar to the one in section 3.2.4, except that it has no support for backtracking. It comes with the following methods,

```
get :: State s s
put :: s → State s ()
```

for reading and writing the state, and the following functions,

```
runState  :: State s α → s → (α,s)
evalState :: State s α → s → α
execState :: State s α → s → s
```

for running the computations.

Here is a small example of a stateful computation:

```
tick = do
    n ← get
    put (n+1)

count40 = replicateM_ 40 tick
```

Here, `tick` increases the state by one, and `count` performs `tick` fourty times. Running at the GHCi prompt gives (using `10` as starting state):

```
*Main> execState count40 10
50
```

6.3.2 Reader

State allows us to both read and write the state. It is often the case that we want to have state that can just be read or just be written. Such monads are provided by the (*Control.Monad.*) *Reader* and *Writer* modules.

A *Reader* computation has access to a *constant state* (often called *environment*), which can be seen as a hidden parameter to the computation. It is defined just like *State*, except that the function does not return a modified state:

```
newtype Reader r a = Reader (r → a)
```

It comes with the method

```
ask :: Reader r r
```

which corresponds to `get` above. Instead of the `put` method (which changes the state for *subsequent* computations), we have the function

```
local :: (r → r) → Reader r a → Reader r a
```

which just runs a local computation in modified environment. For convenience, there is also a function,

```
asks :: (r → a) → Reader r a
```

which simply applies its argument function to the result of `ask`.

Here is a small example where *Reader* is used to give computations access to a variable substitution:

```
add a b = do
    Just x ← asks (lookup a)
    Just y ← asks (lookup b)
    return (x+y)

add2 = do
    x ← local (("a",100):) $ add "a" "b"
    y ← add "a" "b"
    return (x,y)
```

The `add` function looks up its arguments in the environment and adds their results. `add2` runs `add "a" "b"` twice, but the first occurrence gets a modified environment (`"a"` is remapped to `100`). Running the computation,

```
*Main> runReader add2 [("a",10), ("b",20), ("c",30)]
(120,30)
```

shows that the first `add` returned `100+20`, while the second `add` returned `10+20`.

6.3.3 Writer

A *Writer* computation provides the opposite to *Reader*: It can write to the state but not read it:

```
newtype Writer w α = Writer (α,w)
```

The method

```
tell :: Monoid w ⇒ w → Writer w ()
```

corresponds slightly to `put` above. In order for this method to be useful, it needs to be able to "collect" the written values in some way. This functionality is provided by the *Monoid* class. In this thesis, we will only use lists as the `w` parameter, and in that case, collecting is done by the append operator `++`.

In the following example, we define a tree type and a function for summing the node values in the tree:

```
data Tree = Node Int [Tree]

sumNodes :: Tree → Int
sumNodes = sum . execWriter . listNodes
  where
    listNodes (Node n ts) = tell [n] >> mapM_ listNodes ts
```

Here, we use the *Writer* effect to collect all node values in a list, and then the `sum` function for lists to sum them up. Just like `execState`, `execWriter` is a run function which discards the returned value (which in this case is `()`) and only returns the collected writes. Running the function on the following tree,

```
tree =
    Node 3
      [ Node 4
          [ Node 5 []
          , Node 6 []
          , Node 7 []
          ]
      , Node 8 []
      , Node 9 []
      ]
```

at the GHCi prompt, gives:

```
*Main> sumNodes tree
42
```

The *Writer* module also provides the opposite to `local` above:

```
censor :: Monoid w ⇒ (w → w) → Writer w α → Writer w α
```

This method modifies the side effect of a local computation.

```
ask   :: MonadReader r m ⇒ m r
asks  :: MonadReader r m ⇒ (r → α) → m α
local :: MonadReader r m ⇒ (r → r) → m α → m α

tell   :: MonadWriter w m ⇒ w → m ()
censor :: MonadWriter w m ⇒ (w → w) → m α → m α

put :: MonadState s m ⇒ s → m ()
get :: MonadState s m ⇒ m s
```

Figure 6.2: General types of the *Reader*, *Writer* and *State* methods

6.3.4 Monad transformers

It is often needed to have more complex monadic effects than the ones described above. Haskell's standard libraries offer ways to build custom monads by combining the effects of simpler monads [Jon95]. For example, if we want to add *Writer* features on top of an existing monad *M*, we can do this using the *WriterT* *monad transformer* in the following way:

```
type WriterAndM w α = WriterT w M α
```

We see that *WriterT* takes the monad *M* as a parameter, and *M* is then called the *inner* monad. To get the terminology straight, we say that the type constructor *WriterT* `w` is a monad *transformer*, but after application to *M*, we get *WriterT* `w` *M*, which is a *monad*. So the above type definition defines a new custom monad which has the features of both *M* and *Writer*.

Using this technique, we can stack an arbitrary number of monads on top of each other. For example,

```
type MyMonad α = ReaderT Bool (WriterT [Int] (State Double)) α
```

is a monad for computations which can read a *Bool*, write a list of *Int*s and have access to a *Double* state. The problem now arises how to use methods like `put`, `get`, `ask` and `tell` in such a compound monad. This is solved by introducing type classes for each type of effect. For example, *MonadState* contains the methods of the *State* effect. It is instantiated for *State* and *StateT*, but also for monads which have *State* or *StateT* as inner monads. We will not go into the exact details of this overloading – it suffices to say that under normal circumstances, any combination *Reader*/*ReaderT*, *Writer*/*WriterT* and *State*/*StateT* will support methods for all involved effects. The general types of the previously explained methods are listed in figure 6.2.

For a concrete example of using monad transformers, imagine that we want to trace the execution of `add2` from section 6.3.2, by collecting messages as a

Writer effect. To do this, we can simply insert appropriate calls to `tell` in the code of `add`:

```
add a b = do
    Just x ← asks (lookup a)
    tell [a ++ " has value " ++ show x]
    Just y ← asks (lookup b)
    tell [b ++ " has value " ++ show y]
    return (x+y)
```

The definition of `add2` remains the same, but now it has the general type:

```
add2
    :: (MonadReader [(String,Int)] m, MonadWriter [String] m)
    ⇒ m (Int,Int)
```

Two possible instances of this type are

```
add2 :: ReaderT [(String,Int)] (Writer [String]) (Int,Int)
```

```
add2 :: WriterT [String] (Reader [(String,Int)]) (Int,Int)
```

One way to run `add2` is to first run the *Reader* effect and then the *Writer* effect:

```
*Main> let subst = [("a",10), ("b",20), ("c",30)]
*Main> putStr $ unlines $ execWriter $ runReaderT add2 subst
a has value 100
b has value 20
a has value 10
b has value 20
```

This uses the first of the two types above, but we could just as well have done it the other way around, with the same result.

Chapter 7

WiredLava

In this chapter, we develop a replacement of relational Lava from section 4.1. The first part (section 7.1) presents two simple libraries for logical variables which will be used instead of the LP library (chapter 3). The second part (section 7.2) presents a complete Lava implementation to be used in the implementation of Wired (chapter 9). The main improvement over relational Lava is that the logic programming features are now added optionally as a separate layer on top of Lava.

7.1 Light-weight logical variables

In the LP library (section 3.2.5), we represented logical variables by a *substitution* provided through a state monad. Running a logical program then corresponds to a series of updates to this substitution in order to reach a solution. Of course, from a logical perspective, it doesn't make sense to "upadate" a variable – logical variables represent constant facts (or sets of facts in the presence of non-determinism). Thankfully, the library is carfully designed to maintain the illusion of logical variables as constant facts, and the updates just happen under the hood.

Take the following program as an example:

```
logProg = do
    a   ← free
    b   ← free
    a === b
    b === int 5
    return a
```

This program has the following sequence of intermediate substitutions (after running each line):

$$\begin{array}{l}[\mathtt{a} \mapsto \bot] \\ [\mathtt{a} \mapsto \bot, \quad \mathtt{b} \mapsto \bot] \\ [\mathtt{a} \mapsto \mathtt{b}, \quad \mathtt{b} \mapsto \bot] \\ [\mathtt{a} \mapsto \mathtt{b}, \quad \mathtt{b} \mapsto 5]\end{array}$$

That is, it starts with an empty substitution, and gradually transforms it into a solution. The root of `a` is `b`, so running the program will result in the value `5`.

We can think of the lines in the do-block as being combined using `>>`, which corresponds to logical conjunction. With that in mind, it should of course not matter if we swap the two lines containing `===`. That would result in the following sequence of substitutions instead:

$$\begin{array}{l}[\mathtt{a} \mapsto \bot] \\ [\mathtt{a} \mapsto \bot, \quad \mathtt{b} \mapsto \bot] \\ [\mathtt{a} \mapsto \bot, \quad \mathtt{b} \mapsto 5] \\ [\mathtt{a} \mapsto \mathtt{b}, \quad \mathtt{b} \mapsto 5]\end{array}$$

This looks slightly different, but in both cases, the result of running `logProg` is `5`. If, for example, the user were allowed to inspect the current substitution, the property that `>>` is commutative would no longer hold.

7.1.1 The `Knot` monad

So, if logical variables are constant facts, wouldn't it be nice if we could represent them as just that? Instead of having a current substitution, we would then just have a *solution*:

```
type Solution i x = Map i x
```

A solution would be obtained from a set of constraints in the following way

```
type Constraint i x = (i,x)

solve :: Ord i ⇒ [Constraint i x] → Solution i x
solve = fromList
```

A constraint `(i,x)` simply says that index `i` has the value `x`, and there is already a function, `fromList` [Hoo], to create a *Map* from a list of such constraints. The solution from `solve` will only be valid if there are no conflicting constraints.

Now the tricky part is to define computations over these types. The problem is that we need to be able to both declare constraints and read their solution at the same time! Let us ignore this problem for now and start by defining a suitable monad:

```
type Knot i x α = ReaderT (Solution i x) (Writer [Constraint i x]) α
```

(The choice of the name *Knot* will be explained shortly.) This is a simple combination of the *Reader* and *Writer* monads (section 6.3). Computations in this monad will have the ability to declare constraints through *Writer* and read a solution through *Reader*. Note that so far, there is nothing that says that the solution fulfills the constraints – they could be completely unrelated.

Constraints can now be declared using the operator

```
(*=) :: i → x → Knot i x ()
i *= x = tell [(i,x)]
```

and the solution to individual indices can be obtained using

```
askKnot :: Ord i ⇒ i → Knot i x x
askKnot i = asks (! i)
```

A rough transcription of `logProg` to the *Knot* library might look as follows:

```
logProg = do
    a   ←  askKnot "a"
    b   ←  askKnot "b"
    "b" *= a
    "a" *= 5
    return b
```

We start by asking for the solution to the indices `"a"` and `"b"`, then we declare that `"b"` has the value `a` and `"a"` has the value `5`. Note that there is no quantification here – the indices `"a"` and `"b"` can be thought of as globally quantified variables. Another difference to the LP program is that we have to "ask" for the values of variables before using them. Both of these oddities will be handled in section 7.1.2.

Now the only remaining part is to tie the constraints and the solution together:

```
tieKnot :: Ord i ⇒ Knot i x α → (α, Solution i x)
tieKnot knot = (a,solution)
  where
    (a,constraints) = runWriter $ runReaderT knot solution
    solution        = solve constraints
```

This simply runs the two monads in such a way that the solution given to the *Reader* is based on the constraints from the *Writer*. Note that this is a *circular* program – the two local equations have a cyclic dependency. Luckily, Haskell's lazy evaluation semantics makes this work just fine. Here is the result of running `logProg`:

```
*Main> tieKnot logProg
(5, fromList [("a",5),("b",5)])
```

The use of a feedback loop such as in `tieKnot` is sometimes referred to as "tying a knot" [Erk02]. Of course, this is the reason for the naming of our monad. An early paper about circular programming – which is what this is all about – is Richard Bird's [Bir84].

Even though `logProg` appears to be a sequential program, this is only an illusion. Lazy evaluation can be thought of as starting from the result and tracing the data dependencies backwards in order to produce an answer. In particular, expressions are only evaluated *when and if they are needed*. The key observation in this case is the fact that `solve` doesn't need to evaluate its argument completely in order to return the map – it just forces the list constructors and the indices. This means that it is possible to lookup an index without evaluating any elements, so it is perfectly fine for elements in the map to depend on each other. For example, in `logProg`, the element at index `"b"` depends on the element at index `"a"`.

Extended interface

It is often desirable to use *Knot* on top of some other monadic computation, so we also need to define a transformer version of the monad: *KnotT*. To obtain a unified interface to *Knot* and *KnotT*, we use a type class:

```
class ... ⇒ MonadKnot i x m | m → i x
  where
    askKnot    :: i → m x
    askKnotDef :: x → i → m x
    (*=)       :: i → x → m ()
```

This class generalizes *Knot* and *KnotT* in much the same way as, for example, *MonadWriter* generalizes *Writer* and *WriterT*. For example, two possible types of `askKnot` are

```
askKnot :: Ord i ⇒ i → Knot i x x

askKnot :: Ord i ⇒ i → KnotT i x Maybe x
```

The additional method `askKnotDef` has a default parameter which is returned when asking for indices which have no constraints declared on them. The whole library is given in appendix B, and the exposed interface is listed in figure 7.1.

The complete library makes the exposed types abstract in order to protect the logical abstraction. It also relies on various GHC-specific extensions (newtype deriving and undecidable instances) in order to avoid code replication. For example, the following instance includes both *Knot* and *KnotT* at once:

```
instance (Ord i, MonadReader (Map i x) m, MonadWriter [(i,x)] m) ⇒
         MonadKnot i x m
  where
    askKnot i        = asks (! i)
    askKnotDef def i = asks (findWithDefault def i)
    i *= x           = tell [(i,x)]
```

```
data Knot
data KnotT

instance                      Monad         (Knot i x)
instance Ord i ⇒              MonadKnot i x (Knot i x)
instance Monad m ⇒            Monad         (KnotT i x m)
instance                      MonadTrans    (KnotT i x)
instance (Ord i, Monad m) ⇒ MonadKnot i x (KnotT i x m)

askKnot    :: MonadKnot i x m ⇒ i → m x
askKnotDef :: MonadKnot i x m ⇒ x → i → m x
(*=)       :: MonadKnot i x m ⇒ i → x → m ()

accKnot :: Ord i ⇒ (x → x → x) → Knot i x α → (α, Map i x)
tieKnot :: Ord i ⇒               Knot i x α → (α, Map i x)

accKnotT :: (Ord i, MonadFix m)
         ⇒ (x → x → x) → KnotT i x m α → m (α, Map i x)

tieKnotT :: (Ord i, MonadFix m) ⇒ KnotT i x m α → m (α, Map i x)
```

Figure 7.1: The *Knot* interface

Constraint resolution

The `solve` function used in `tieKnot` assumes the absence of conflicting constraints. The following, alternative run function lets the user supply a custom function for resolving constraints:

```
accKnot :: Ord i ⇒ (x → x → x) → Knot i x α → (α, Map i x)
accKnot acc knot = (a,solution)
  where
    (a,constraints) = runWriter $ runReaderT knot solution
    solution        = fromListWith acc constraints
```

The *accumulator* `acc` is used to combine multiple contraints on the same index. In general, constraints should not be sensitive to their internal order or the structure of the program; therefore, it normally only makes sense to use commutative and associative accumulators. The ability to use a custom accumulator goes beyond the functionality offered by the LP library (which always unifies multiple constraints), and this feature will prove to be very useful when defining different circuit analyses in section 7.2.4.

In case there is no sensible accumulator available, we can always avoid multiple constraints by throwing an error. A safer definition of `tieKnot` would then be

```
tieKnot = accKnot (error "Over-constrained!")
```

Here is an example run:

```
*Main> tieKnot ("a"*=5 >> "a"*=4)
((),fromList [("a",*** Exception: Over-constrained!
```

7.1.2 The `Let` monad

The *Knot* library used explicit indices to the underlying mapping to express logical-like computations. In order to view these indices as logical variables, we have to think of them as being globally quantified, and of course this limits the kind of computations that we can express. By adding a simple abstraction on top of *Knot*, we can get something which much more resembles the LP library.

The extended monad, called *Let*, uses integers as index and a state monad to thread a fresh index through the computation:

```
type VarId = Int

type Let x a = StateT VarId (Knot VarId x) a
```

Instead of using *VarId* directly for variables, we are going to pair each index together with its value:

```
data Var x = Var VarId x
```

Creating a free variable is now done as

```
free = do
    vid ← get
    put (succ vid)
    x ← askKnot vid
    return (Var vid x)
```

This function gets a free index, increases the index counter and returns the index paired with the value obtained from `askKnot`. By letting the variable include its value, we can now use a *pure* function to extract it:[1]

```
val :: Var x → x
val (Var _ x) = x
```

It makes very much sense to do this as a pure function, because altough `askKnot` is monadic, it will always yield the same value for the same index.

Now the only missing functions are for declaring constraints and running the computations:

```
(===) :: Var x → x → Let x ()
Var vid _ === x = vid *= x

runLet :: Let x a → a
runLet ma = fst $ fst $ tieKnot $ runStateT ma 0
```

[1] Thanks to Matthew Naylor for this idea, and for the nice naming of the *Let* monad.

Getting back to the previous `logProg` example, the same program using *`Let`* is almost identical to the LP version:

```
logProg = do
    a   ← free
    b   ← free
    a === val b
    b === 5
    return (val a)
```

The naming of the library comes from the fact that *`Let`* allows us to introduce variables in a similar way to **`let`** expressions. Note, for example, the similarities between the following two programs:

```
testLet = runLet $ do
    ones   ← free
    ones === '1':val ones
    return $ take 10 (val ones)

testLet = let
    ones = '1':ones
    in take 10 ones
```

Running either program yields the result

```
*Main> testLet
"1111111111"
```

However, *`Let`* is in a way more powerful, since it treats variables as first-class objects. For example, in the `buildCirc` function from section 4.1.1, we started by creating a table of free variables and referred to those when building the circuit. This is of course only possible if variables can be treated as values. We will now show how to define `buildCirc` using *`Let`*.

Signal flow abstraction

In section 4.1.1, we looked at how to construct a circuit from a flat netlist description. Using a special "relational Lava", implemented in the LP library, we were able to get a concise declarative solution, even without any assumptions about the order between components in the netlist. However, this required a completely new (although similar) replacement of Lava in order for signals to support logical operations.

Figure 7.2 shows the very minor changes needed in order to achieve the same result using *`Let`* and the *standard* Lava library. Here, *`Signal Bool`* is the normal Lava signal type (section 4.1.2). The code in `declareSigs` can be resued as it is if we just drop the type signature.

```
...

data Comp = Comp
       { gate :: [Signal Bool] → Signal Bool
       , inps :: [Position]
       , out  :: Position
       }

...

buildCirc circ ss = do
    posMap  ← declareSigs circ
    let pos = (posMap !)
    sequence_ [ pos i === s | (i,s) ← zip (inputs circ) ss ]
    mapM_ (buildComp pos) (comps circ)
    return (map (val.pos) (outputs circ))
  where
    buildComp pos (Comp gt is o) = pos o === gt (map (val.pos) is)
```

Figure 7.2: Redefinition of `buildCirc` using *Let*

Extended interface

The complete *Let* library is given in appendix C. Just like for the *Knot* library, we have a transformer version (*LetT*) and the types are exported abstractly. The interface is summarized in figure 7.3.

Comparison with LP

So far, we have pointed at the similarities between the *Let* and LP libraries, but there are also some differences.

In some ways, *Let* is less general than LP:

- *Let* only supports constraints of the form `variable === value`, and such constraints have a definite right-to-left flow of information. In contrast, unification in LP works on arbitrary nested terms and has abstract information flow.
- *Let*, as presented here, requires all variables to be of the same type.[2]
- *Let* has no support for non-determinism. Since constraints in *Let* are solved globally for the whole computation, implementing backtracking on individual variables appears to be non-trivial.

[2]There appears to be ways around this using unsafe type casting, but we will not go into any details of that here.

```
data Let x α
data LetT x m α

instance             Monad      (Let x)
instance             MonadLet x (Let x)
instance Monad m ⇒ Monad      (LetT x m)
instance             MonadTrans (LetT x)
instance Monad m ⇒ MonadLet x (LetT x m)

data Var x

free  :: MonadLet x m ⇒ m (Var x)
(===) :: MonadLet x m ⇒ Var x → x → m ()
val   :: Var x → x

runLet :: Let x α → α

runLetT :: MonadFix m ⇒ LetT x m α → m α
```

Figure 7.3: The *Let* interface

Interestingly, none of these shortcomings is very important for our application. For example, unification of two signals usually only makes sense if one is a free variable – most other cases represent some kind of short circuit.

There are also advantages of *Let* compared to LP:

- *Let* has a much simpler implementation.
- *Let* allows a more modular design. For example, it can be used on top of standard Lava rather than requiring a completely new circuit library.
- *Let* trivially supports feedback loops (which were tricky to get right with LP, see section 4.1.2)

7.2 WiredLava

As demonstrated in the previous section, signal flow abstraction can be obtained using functionality from the *Let* library on top of Lava. In order to get a simple presentation, we will now define a new version of Lava, called *WiredLava*. But do not worry! This is the last circuit library we will see, and it is actually used in the current implementation of Wired (chapter 9). WiredLava is also more flexible than the normal Lava in that it supports primitive components with multiple outputs. However, Lava is currently being revised, and in the future, it should be possible to use the ordinary Lava library inside Wired. Until then,

we can safely say that WiredLava is *a* version of Lava, so it is perfectly valid to view Wired as an extension to Lava.

The complete WiredLava library is given in appendix D. In this chapter, we will zoom in on some of the more interesting parts.

7.2.1 Signals

The *Signal* type is defined as

```
data Node cell s
       = Constant Name
       | Cell CellPin cell NumOuts [s]

data Signal cell = Signal NodeId (Node cell (Signal cell))
```

This is similar to the definition from section 4.1.2, except that there are only two possible nodes – cells and constants. Different types of cells (for example, AND-gates and OR-gates) can be distinguished using the `cell` parameter, which can be of any type. A cell node also contains information about which output it represents (*CellPin*), in case the cell has multiple outputs, and the cell's total number of outputs (*NumOuts*). The types *CellPin*, *NumOuts* and *NodeId* are aliases for *Int*.

In order to supply unique node identifiers to signals, we use a simple state monad:[3]

```
type Lava = State NodeId
```

The following two invariants are expected to hold for values of the *Signal* type:

- If two sub-trees have the same *NodeId*, they may differ only in the *CellPin* number. That is, sub-trees with the same *NodeId* can be seen as sharing the same physical cell.
- The *CellPin* number of a cell with `n` outputs must be between `0` and `n-1`.

The examples in this chapter will use the following concrete signal type:

```
data Cells = Inv | And2 | Or2 | HalfAdd
type Sig   = Signal Cells
```

7.2.2 Ports

Signal is the basic type for circuits. In order to get a more useful library, we want to support arbitrary signal structures as interfaces to circuits. For example, the `and2` gate has type

```
and2 :: (Sig,Sig) → Lava Sig
```

[3] In the real implementation, *Lava* is an abstract data type.

and a binary adder might have the type

```
adder :: (Sig,([Sig],[Sig])) → Lava ([Sig],Sig)
```

Signal structures are called *ports*, and the *Port* classes (*Port*, *PortStruct* and *PortFixed*) allow us to treat these structures in a unified way. For example, `mapPort` is used to map a function over a port in such a way that the structure is preserved:

```
mapPort :: (PortStruct pa sa t, PortStruct pb sb t)
        ⇒ (sa → sb) → (pa → pb)
```

Here, `pa` is a port whose leaves have type `sa`, and similarly for `pb` and `sb`. So, given a function to convert between leaves, `mapPort` returns a function to convert between structures of leaves. The last parameter of *PortStruct* is a canonical representation of the structure (independent of the leaf type), and by using the same parameter `t` for both `pa` and `pb`, we ensure that we can only map between matching structures. Interestingly, the code would still type check if we didn't use the same `t`, but then we would instead get run-time errors when trying to convert between mismatching structures. It should, however, be noted that `mapPort` only ensures type-level matching – properties that are not reflected in the type, such as list lengths, are not checked.

For instance, `mapPort` can be used as follows:

```
*Main> mapPort even ((1::Int,2::Int),[3::Int,4])
((False,True),[False,True])
```

This simple example converted a structure of *Int*s to a matching structure of *Bool*s. The same idea is used when analyzing circuits. For example, the `simulate` function, which we will define later, has the type

```
simulate
    :: ( PortStruct pbi Bool ti
       , PortStruct psi Sig  ti
       , PortStruct pso Sig  to
       , PortStruct pbo Bool to
       )
    ⇒ (psi → Lava pso) → (pbi → pbo)
```

This simply converts a function on *Sig*s to a function on *Bool*s, while ensuring that the structures remain intact.

The *Port* class corresponds to *Generic* in Lava [Cla00].

7.2.3 Cell libraries

A cell library can easily be created using the combinators `constant` and `cell`, as shown in figure 7.4. Here it is important to give type signatures, since that is the only way to let `cell` know how many outputs each cell has.

```
low, high :: Sig

low  = constant 0 "low"
high = constant 1 "high"

inv :: Sig → Lava Sig
inv = cell Inv

and2 :: (Sig,Sig) → Lava Sig
and2 = cell And2

or2 :: (Sig,Sig) → Lava Sig
or2 = cell Or2

halfAdd :: (Sig,Sig) → Lava (Sig,Sig)
halfAdd = cell HalfAdd
```

Figure 7.4: A simple cell library

```
circ1 a = do
    b ← inv a
    and2 (b,b)

circ2 = and2    >=> inv
circ3 = halfAdd >=> and2

circ4 a = do
    b ← and2 (a,a)
    c ← or2  (b,b)
    d ← and2 (b,c)
    return (b,d)
```

Figure 7.5: Example circuits

Figure 7.5 defines a few simple example circuits. To see what the constructed circuits actually look like, we can simply run them at the GHCi prompt:[4]

```
*Main> evalState circ1 2

Signal 3 (Cell 0 And2 1
  [ Signal 2 (Cell 0 Inv 1
      [ Signal 0 (Constant "low")
      ])
  , Signal 2 (Cell 0 Inv 1
      [ Signal 0 (Constant "low")
      ])
  ])
```

Note that although the tree expansion contains two *Inv* cells, both have the node *id* 2, so any clever analysis should be able to treat them as a single cell.

7.2.4 Interpretation

So far, we have seen how to construct circuits in WiredLava. Now we will look at how to analyze the resulting networks. The *Interpretation* module in appendix E provides the necessary machinery. An interpretation is defined by the following type:

```
data Interpretation cell x = Interp
       { constants   :: [x]
       , defaultVal  :: x
       , accumulator :: x → x → x
       , propagator  :: cell → ([x] → [Maybe x])
       }
```

The type is parameterized on the cell library `cell` and the interpreted signal values `x`. For example, to define Boolean simulation for the library in figure 7.4, we would use the type *Interpretation Cells Bool*.

The `constants` field gives the interpretation of each constant; the first element represents the *id* `0`, the second element represents the *id* `1`, and so on (see the numbers given to `constant` in figure 7.4). The field `defaultVal` gives the interpretation of signals that are not constrained by the analysis, and `accumulator` defines how to combine multiple constraints on the same signal.

Finally, the `propagator` field defines a propagation function for each cell. This function receives a list of all the cell's signals – first inputs, then outputs – and returns constraints for the very same signals. The use of *Maybe* in the result type allows us to leave certain signals unconstrained. This function has a completely abstract signal flow – information can be propagated from inputs to outputs, from outputs to inputs, or in both directions at once.

[4] We start the enumeration with the node *id* 2, since `0` and `1` are reserved for `low` and `high` (in this particular library).

```
interpBool :: Interpretation Cells Bool
interpBool = Interp
    { constants  = [False,True]
    , propagator = prop
    }
  where
    prop Inv     = λ[a,_]   → [Nothing, Just (¬ a)]
    prop And2    = λ[a,b,_] → [Nothing, Nothing, Just (a ∧ b)]
    prop Or2     = λ[a,b,_] → [Nothing, Nothing, Just (a ∨ b)]
    prop HalfAdd =
        λ[a,b,_,_] → [Nothing, Nothing, Just (a ≠ b), Just (a ∧ b)]
```

Figure 7.6: Boolean interpretation of cell library

Boolean simulation

A Boolean interpretation of our small cell library (figure 7.4) is given in figure 7.6. This interpretation has only *forwards* information flow, which can be seen from the fact that the anonymous functions ignore the output signals in their arguments, and return *Nothing* for the input signals in their results. Note that we omitted the `defaultVal` and `accumulator` fields here. There is simply no way we can end up with under- or overconstrained signals in the given interpretation.

The actual simulation function can now be defined as

```
simulate = interpret interpBool
```

and this can now be used to simulate our example circuits:

```
*Main> simulate circ1 False
True

*Main> simulate circ2 (False,True)
True

*Main> simulate circ3 (False,True)
False
```

Fanout analysis

For an example of a different interpretation, we demonstrate a fanout analysis, defined in figure 7.7. This analysis has a *backwards* information flow – the cells propagate the value `1` to each input – and it uses `(+)` as accumulator in order to count all ones that are assigned to a given signal. The previously defined `circ4` exports the signal `b`, which is used three times as input to other gates. Fanout analysis of the circuit confirms this observation:

```
interpFanout :: Interpretation Cells Int
interpFanout = Interp
    { constants   = [0,0]
    , defaultVal  = 0
    , accumulator = (+)
    , propagator  = prop
    }
  where
    prop Inv     = const [Just 1,          Nothing]
    prop And2    = const [Just 1, Just 1, Nothing]
    prop Or2     = const [Just 1, Just 1, Nothing]
    prop HalfAdd = const [Just 1, Just 1, Nothing, Nothing]
```

Figure 7.7: Fanout interpretation of cell library

```
*Main> interpret interpFanout circ4 0
(3,0)
```

Interpretation through a type class

Our approach uses *external* interpretation of circuits – that is, we first construct the signal tree, and then we interpret it. An alternative approch, used in Hydra [O'D95, O'D04], is to instead provide the interpretations through a type class. That way, gates bahave differently depending on their type. The following circuit is defined in [O'D04]:

```
circ :: Signal α ⇒ α → α → α → α
circ a b c = x
  where
    x = or2 (and2 a b) (inv c)
```

By providing inputs of different types, this circuit will behave quite differently:

```
*Main> circ False False False
True

*Main> circ (Inport "a") (Inport "b") (Inport "c")
Or2 (And2 (Inport "a") (Inport "b")) (Inv (Inport "c"))
```

In the first run, the circuit *is* a boolean function which performs the simulation without any external interpretation. The second run uses an interpretation where each gate constructs a node in a tree. Circuits in WiredLava always behave according to the second interpretation.

The Hydra solution is very clean and powerful, and makes it easy to add new interpretations. But it has the drawback that all gates have to be methods of the *Signal* class, and thus it is not possible to extend the library with new

gates. Our *Interpretation* type is parameterized on both the cell library and the interpretation, and is therefore not limited in the same way.

Implementation

The implementation of `interpret` is given in appendix E. The real work is done by the function `interpretSig`, which traverses the signal graph and performs the actual propagations. It uses the *Traversal* monad from appendix F, which is a combination of *KnotT* and *Visit*:

```
newtype Traversal s x a =
          Traversal { unTraversal :: KnotT Pin x (Visit s) a }
    deriving (Monad, MonadReader (Map Pin x), MonadWriter [(Pin,x)])
```

The *Pin* type uniquely identifies a signal in the circuit. *KnotT* takes care of associating each *Pin* with an interpretation value `x`, and *Visit* is used to keep track of which cells have already been visited[5]. The simplicity of the `interpret` function is largely due to these helper monads. Thanks to data flow abstraction through *KnotT*, a single traversal of the signal graph is sufficient, even with bidirectional interpretations, and thanks to *Visit*, we are sure not to interpret the same cell more than once.

It is very interesting to note the similarity between the propagation function in the *Interpretation* type and the *Relation* type from section 2.4. Apart from the fact that the *Relation* type encoded both geometrical and structural constraints, they are conceptually the same thing. However, a big difference between the two implementations is that the "instantiation engine" used a very naive and inefficient fixpoint iteration. The implementation in this section does the whole interpretation in a single traversal!

7.3 Discussion

This chapter has presented a simple form of data flow abstraction, relying on Haskell's lazy evaluation semantics rather than supplementary evaluation strategies as in chapters 2 and 3. The main functionality is provided by the *Knot* library, which has a very simple implementation. A small extension to this library resulted in the *Let* library, which can be used to achieve signal flow abstraction in a hardware description library such as Lava. The most important difference compared to "relational Lava" from section 4.1 is that *Let* can be used on top of *existing* libraries in a modular way. Relational Lava, on the other hand, had to be specially-taylored to work with the LP library, and this resulted in a more complicated impementation.

[5] For a more "pure" solution to the "visited" problem, see [Joh98].

In addition, the *Knot* library was used to implement a general interpretation function for Lava circuits. Once again, data flow abstraction through *Knot* allowed for a very simple, yet powerful, implementation.

The *Knot*/*Let* approach to logical variables is much more light-weight than LP. It is also more restricted (for example, no general unification), but this is not important for current Wired. Futhermore, there is one aspect in which *Knot* is more powerful than LP: It supports custom accumulation of constraints. This is what made the fanout analysis in section 7.2.4 so simple. The similar analysis in section 3.4.3 had to use an explicit fork to achieve the same kind of backwards propagation, and that would not be acceptable for a library like WiredLava.

Chapter 8

Layout

The initial example of a circuit in Wired (chapter 6) illustrated how data flow and placement have been separated to enable a more flexible system. The key to this separation is a modular implementation where structure and geometry are handled by separate stand-alone libraries. The structural representation has already been developed in chapter 7. We will now define the library that handles geometry. Section 8.1 shows how the geometry is represented, and section 8.2 gives a monadic interface to computations with layout.

8.1 Representing floorplans

In section 1.2.3, we mentioned a "slicing tree" representation of floorplans – essentially a tree where each node is annotated by the relative placement of its children. We use a Haskell data type to represent such trees. The full implementation is given in appendix G, and the *Floorplan* type looks as follows:

```
data Floorplan bl
       = Block Width Height bl
       | Comb Placement [Floorplan bl]
```

That is, a *Floorplan* is either a primitive block with a certain width and height or a combination of other floorplans with a given relative placement. Each block has an extra argument of type `bl` which makes is possible to store custom information in the blocks.

Relative placement is defined as

```
data Placement
       = Unspecified
       | Merged Alignment {- x -} Alignment {- y -}
       | Row Direction Alignment

data Direction = Rightwards | Leftwards | Upwards | Downwards
```

Unspecified should be self-explanatory, *Merged* placement means that the sub-floorplans are stacked on the same position, and *Row* means that they are placed in order next to each other. The *Alignment* argument specifies how to line up blocks with unmatching width/height. An alignment of 0 means towards the bottom-left corner and 1 means towards top-right.

When analyzing floorplans, blocks are treated either as free space or as boxes with names. This can be modeled using the *Maybe* type:

```
type BlockType = Maybe Name
```

It is possible to have more information in the blocks, but it should at least contain this minimum info. The *IsBlock* class provides a general interface to such block types.

8.1.1 Generating postscript

One can imagine several different back-ends for the *Floorplan* type. In section 9.3, we generate output for a tool for physical circuit design, and here we give a function for generating a postscript picture:

```
renderFloorplan_ :: IsBlock bl ⇒
    String → Floorplan bl → [(Position,Position)] → IO ()
```

This function takes a file name, a floorplan and a list of lines (wires), and writes the corresponding postscript to the given file. Since we do not yet have any notion of wires, we can also use the simpler function:

```
renderFloorplan :: IsBlock bl ⇒ String → Floorplan bl → IO ()
renderFloorplan title fp = renderFloorplan_ title fp []
```

An example floorplan, `fp`, which uses the minimal block type, is shown in figure 8.1. The result of running

```
*Main> renderFloorplan "fp" fp
```

is shown in figure 8.2. Note how the left column is aligned towards the left border (alignment `0`) while the right column is aligned towards the right border (alignment `1`).

The `renderFloorplan` function is based on a more general function for listing the absolute positions of blocks in the floorplan:

```
listAbsolute :: IsBlock bl ⇒
    Floorplan bl → [(bl, (Width,Height), (Width,Height))]
```

```
fp :: Floorplan BlockType
fp =
  Comb (Row Rightwards 0)
    [ Comb (Row Downwards 0)
        [ Block 20 30 (Just "Block1")
        , Block 40 30 Nothing
        , Block 30 30 (Just "Block2")
        ]
    , Comb (Row Downwards 1)
        [ Block 30 10 Nothing
        , Block 40 40 (Just "Block3")
        , Block 30 10 Nothing
        , Block 30 30 (Just "Block4")
        , Block 30 30 (Just "Block5")
        ]
    ]
```

Figure 8.1: Example floorplan

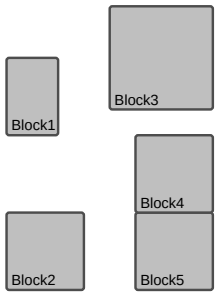
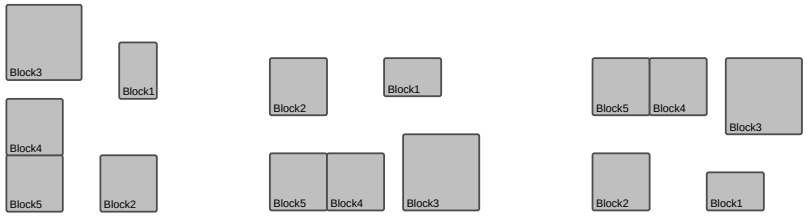


Figure 8.2: `fp` rendered as postscript



(a) (b) (c)

Figure 8.3: Transformations of `fp`: (a) `flipX fp`, (b) `rotate 3 fp`, and (c) `flipY $ rotate 3 fp`

The first `(Width,Height)` pair is the absolute position and the second pair is the block's own size. This function is not technically very challenging – a single tree traversal with the accumulated absolute position as a threaded parameter – but it is large enough that we choose to omit it from the text.

8.1.2 Transformations

There are three basic transformations available for floorplans: `rotate`, `flipX` and `flipY`. Since these operations are quite general, they are overloaded through the *Transformable* class. Figure 8.3 shows three different transformations of the example floorplan. The argument to `rotate` means number of steps (90°) *counter-clockwise*. The transformations are rather trivial to define. The following defines `flipX` for floorplans by recurively flipping the children of each node:

```
flipX (Block w h bl) = Block w h (flipX bl)
flipX (Comb pl fps)  = Comb (flipX pl) (map flipX fps)
```

8.2 Layout as a side-effect

Writing floorplans explicitly, as in figure 8.1, is not very convenient. There are probably different ways in which we could make a simpler interface to floorplans; however, a monadic interface (section 3.2.1) turns out to be particularly advantageous, as it allows us to treat the creation of the floorplan as a *side-effect* of some other computation. Once again, a combination of *Reader* and *Writer* (see section 6.3 and 7.1.1) does the job:

```
newtype Layout bl α = Layout
          (ReaderT Placement (Writer [Floorplan bl]) α)
        deriving
          ( Monad
          , MonadReader Placement
          , MonadWriter [Floorplan bl]
          , MonadFix
          )
```

Let us ignore the *Reader* effect for now. The *Writer* effect allows us to create the floorplan by writing out its sub-blocks as a list. Since this is just a normal list, we can view the code as being generic in terms of its relative placement. We can, however, fix the placement using the following function:

```
subLayout :: Placement → Layout bl α → Layout bl α
subLayout pl m = local (const pl) $ censor (λfps → [Comb pl fps]) m
```

Here, we use `censor` (section 6.3.3) to transform the written floorplan list. The transformation simply creates a new floorplan node with the previously written blocks as children and using the given placement argument `pl`. The result is a

```
Comb Unspecified
  [ Comb (Row Rightwards 0)
      [ Comb (Row Leftwards 0)
          [ Block 30 20 (Just "Block1")
          , Block 20 20 Nothing
          , Block 40 20 (Just "Block2")
          ]
      , Comb (Row Downwards 0)
          [ Block 30 20 (Just "Block1")
          , Block 20 20 Nothing
          , Block 40 20 (Just "Block2")
          ]
      ]
  ]
```

Figure 8.4: Floorplan of the `layout` example

singleton list containing only the new node, so this floorplan can now be used as a sub-block in a larger floorplan with a possibly different placement. The additional application of `local` will be explaind below.

An example of a layout computation is

```
threeBlocks :: Layout BlockType ()
threeBlocks = do
    tell [Block 30 20 (Just "Block1")]
    tell [Block 20 20 Nothing]
    tell [Block 40 20 (Just "Block2")]
```

which can be used as a building block in a larger computation:

```
layout = subLayout (Row Rightwards 0) $ do
    subLayout (Row Leftwards 0) threeBlocks
    subLayout (Row Downwards 0) threeBlocks
```

Using the following run function,

```
runLayout :: Layout bl α → (α, Floorplan bl)
runLayout (Layout m) = (a, Comb Unspecified fps)
  where
    (a,fps) = runWriter $ flip runReaderT Unspecified m
```

we find that `layout` results in the floorplan in figures 8.4 and 8.5. Having *`Unspecified`* placement on the top level is fine since the top node only has one child.

Note how the building block `threeBlocks` is used with two different placements (*`Leftwards`* and *`Downwards`*), while the definition of `threeBlocks` is completely ignorant of its placement. This can be compared to the approach in chapter 2

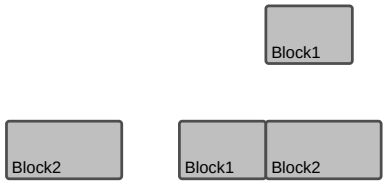


Figure 8.5: Postscript floorplan of the `layout` example

in which all combinators had a pre-defined direction (for example, rightwards for `*||*` and upwards for `*==*`). In fact, we can view our new library as having a *single* layout combinator, `>>=`, with generic relative placement.

8.2.1 Combinators

An obvious way to simplify the example above is to define shorthands for the `subLayout` calls:

```
rightwards al = subLayout (Row Rightwards al)
leftwards  al = subLayout (Row Leftwards  al)
upwards    al = subLayout (Row Upwards    al)
downwards  al = subLayout (Row Downwards  al)
```

These should be recognized as the layout "annotations" from section 6.2. Block creation can also be made simpler:

```
block :: Width → Height → bl → α → Layout bl α
block x y bl a = tell [Block x y bl] >> return a
```

This function creates a block and returns the given value. This value represents the actual computation done by the block – the floorplan is just a side-effect.

Space is only used to separate blocks in a placement row. Therefore, space blocks only need to extend in the direction of the relative placement. This is where the ***Reader*** functionality of the ***Layout*** monad comes in – computations can ask the ***Reader*** for the current placement. A thin block that extends only in the placement direction is defined as

```
thinBlock :: bl → Width → α → Layout bl α
thinBlock bl w a = do
    pl ← ask
    case pl of
         Row Rightwards _ → block w 0 bl a
         Row Leftwards  _ → block w 0 bl a
         Row Upwards    _ → block 0 w bl a
         Row Downwards  _ → block 0 w bl a
         _                → return a
```

If the placement is not a *Row*, there is no direction to talk about, so no space is created. The placement environment is kept up-to-date by `subLayout` using the `local` method from the *Reader* library (section 6.3.2). Now, `thinBlock` can be specialized to constructing pure space:

```
space :: IsBlock bl ⇒ Width → α → Layout bl α
space = thinBlock spaceBlock
```

The previous example then simplifies to:

```
threeBlocks = do
    block 30 20 (Just "Block1") ()
    space 20 ()
    block 40 20 (Just "Block2") ()

layout = rightwards 0 $ do
    leftwards 0 threeBlocks
    downwards 0 threeBlocks
```

Since none of the blocks represents any useful computation, we just give `()` as the returned value. Actually, thin blocks, like `space`, are usually not supposed to compute anything, so it may seem strange for them to return a value. However, it often turns out to be very handy to be able to just pass a value through such blocks. It allows for simple point-free definitions, such as `sklanskyIO` in section 9.1.4.

8.2.2 Extended interface

Appendix H contains the full implementation of the *Layout* library. Just like for *Knot* and *Let*, we also provide a transformer version of the monad and use a type class (*MonadLayout*) to obtain a unified interface. The full library also contains a method for performing arbitrary transformations of the written floorplan. This is used to instantiate the monads under the *Transformable* class, so that layout computations can be flipped and rotated just as we could with floorplans.

8.3 Layout of arithmetic computations

The previous example concentrated only on constructing floorplans. However, a very nice feature of the *Layout* library is that it can be used as an extra layer on top of other computations. We will now look at a simple example of how to express the layout of arithmetic computations.

We start by defining a few primitive blocks:

```
box :: Width → Height → Name → α → Layout BlockType α
box w h nm = block w h (Just nm)

add, sub, mul :: Int → Int → Layout BlockType Int

add a b = box 60  30 "+" (a+b)
sub a b = box 50  50 "-" (a-b)
mul a b = box 100 50 "*" (a*b)
```

These can be seen as physical computational units. They can now, for example, be used to define a multiply-accumulate unit:

```
mac acc []          = return acc
mac acc ((a,b):abs) = rightwards 0 $ do
    acc' ← upwards 0 $ add acc =<< mul a b
    mac acc' abs
```

This recursive function takes a list of pairs, multiplies the components of each pair and sums all the resulting products. In order to make the example more interesting, we instantiate two `mac` units and substract their result:

```
arith as bs = rightwards 0.5 $ do
    (a,b) ← downwards 0 $ do
        a ← mac 0 as
        space 20 ()
        b ← mac 0 bs
        return (a,b)
    space 20 ()
    sub b a
```

To "simulate" this computation, we just throw away the floorplan after running the monad:

```
*Main> let as = [(2,2),(3,4),(5,6),(7,8)]
*Main> let bs = [(5,6),(7,8),(9,10),(11,12)]
*Main> fst $ runLayout $ arith as bs
206
```

And if we want to view the resulting picture, we simply extract the floorplan and render it:

```
*Main> let as = [(2,2),(3,4),(5,6),(7,8)]
*Main> let bs = [(5,6),(7,8),(9,10),(11,12)]
*Main> renderFloorplan "arith" $ snd $ runLayout $ arith as bs
```

The result is shown in figure 8.6.

8.4 Discussion

In this chapter, we have developed a general library for expressing the layout of computations. A monadic interface allows us to treat layout generation as a side-

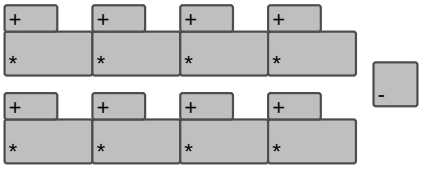


Figure 8.6: Floorplan of the multiply-accumulate example

effect of some other computation. The main computaion is described in standard monadic style, while layout information is supplied as simple annotations.

Even though the main motivation for the library was its use in Wired (chapter 9), it is not in any way limited to standard cell layouts. For example, it might also be usful as a tool for higher-level floorplanning of chip designs, and possibly even for applications beyond digital circuits. In this chapter, we have mainly used it as a way of visualizing computations, but in Wired, the constructed floorplan will also be used to estimate wire lengths and to export the placement to CAD tools.

Chapter 9

Wired

Previous chapters have developed the various building blocks that are needed for the new implementation of Wired. Now it is time to assemble the pieces and build the actual system. Most of the work has already been done, so the new Wired implementation turns out to be quite simple.

9.1 Extending Lava with Layout

The system that was demonstrated in chapter 6 can be implemented simply by combining WiredLava (chapter 7) and the layout library (chapter 8). The idea is very similar to how we added layout on top of arithmetic computations in section 8.3. We start by constructing a custom monad by wrapping *LayoutT* over *Lava*:[1]

```
type Circ = LayoutT WiredBlock Lava
```

The block information *WiredBlock* now needs to contain some more information than the previously used *BlockType* (section 8.1). In addition to knowing whether a floorplan block is a box or empty space, we also need to associate each block with a node *id* from the Lava circuit and to keep track of the cell's orientation (affected by the operations from the *Transformable* class):

[1] In the real implementation, *Circ* is an abstract data type.

```
data WiredBlock
       =  Space
       |  WCell NodeId Name Orientation
       |  Guide Pin Layer

type Orientation = (Bool,Direction)
type Layer       = Int
```

This type also has a third kind of block, *Guide*, which will be explained later. The block interpretation of this type is given by

```
instance IsBlock WiredBlock
  where
    spaceBlock = Space

    toBlock (WCell _ nm _) = Just nm
    toBlock _              = Nothing
```

which says that *WCell* represents a box, and the other two constructors represent empty space.

9.1.1 Lifting cells from Lava

Turning a Lava gate into a Wired cell is done by the function

```
liftGate :: (Port α (Signal cell), Port β (Signal cell)) ⇒
    Name → Width → Height → (α → Lava β) → (α → Circ β)

liftGate nm x y lava a = do
    b ← LayoutT $ lift $ lift $ lava a
    let nid = portNode b
    block x y (WCell nid nm (False,Upwards)) b
  where
    portNode = head . map getNode . toList . port
      where
        getNode (Signal nid _) = nid
```

After applying the inputs, the node *id* `nid` of the first output is obtained and stored in the constructed floorplan block. The function `toList` is a method of the *Foldable* class [Hoo]. The block is also given the standard orientation (*False*,*Upwards*). This function only makes sense if the lifted gate represents *one* node in the signal graph. Otherwise, the circuit will contain nodes which are not present in the floorplan. The idea is, however, that `liftGate` should only be used to create cell libraries. Ordinary users of Wired will just use these libraries, and should not be exposed to functions like `liftGate`.

9.1.2 Running circuits

Applying `runLayoutT` to a Wired circuit corresponds to turning it into a Lava circuit that returns a floorplan on the side:

```
runLayoutT :: Circ a → Lava (a, Floorplan WiredBlock)
```

Since we are often concerned with either just the structural view or just the geometrical view, we define

```
stripLayout :: Circ a → Lava a
stripLayout = liftM fst . runLayoutT

getFloorplan :: NodeId → Circ a → Floorplan WiredBlock
getFloorplan nid = snd . fst . flip runState nid . runLayoutT

renderCircuit :: NodeId → String → Circ a → IO ()
renderCircuit nid title = renderFloorplan title . getFloorplan nid
```

The *NodeId* argument to `renderCircuit` tells where to start the node enumeration in the Lava circuit. This value should be higher than the node *id*:s of all constants in the circuit (see example in section 7.2.3), so its value may differ between different cell libraries. Therefore, the idea is that each cell library should reexport `renderCircuit` with the correct value hard-coded; that is, each cell library should export a function with the type

```
renderCircuit :: String → Circ a → IO ()
```

9.1.3 Primary inputs

Our circuits normally have a type of the form $\alpha \rightarrow Circ\ \beta$, but functions like `renderCircuit` expect the type $Circ\ \alpha$. One way to get this latter form is to apply the circuit to constant values. For example, in section 6.2, we did the following to view the layout of `sklansky`:

```
*Main> renderCircuitNets "skl" $ sklansky dot (replicate 16 low)
```

In general, however, it is not very useful to have a `sklansky` network with all inputs hard-wired to `low`. For example, when exporting the network to a CAD tool (section 9.3), each input signal should be reported as a unique primary input.

A primary input is created by the function

```
inputS :: Name → Circ (Signal cell)
inputS nm = do
    nid ← lift get
    lift $ put (succ nid)
    return $ Signal nid (Constant nm)
```

This function can be generalized to ports of fixed size:[2]

```
input :: ∀ p cell . PortFixed p (Signal cell) ⇒ Name → Circ p
input nm = do
    ss ← replicateM (lengthFP (undefined :: p)) (inputS nm)
    return $ fromListFP ss
```

The `nm` parameter is just a way to label the signal – it is really the node *id* that identifies the signal, so all signals created by this function will be reliably unique. The `input` function will be used when we render a refined version of `sklansky` in the next section.

9.1.4 Routing guides

So far, we have seen how to use layout annotations to express the layout of cells. This may be perfectly adequate, since it gives a good indication of the lengths of individual wires. But in order to get some more control over the routing, we may want to be able to guide the wires between the cells as well. The idea is simple: Instead of associating each signal node with a single block in the floorplan, it may be associated with several blocks. However, exactly *one* of those blocks should be a cell – all others must be guides. The latter are represented by the *Guide* constructor of the *WiredBlock* type, which says that some space in the floorplan should contain the wire from a certain pin routed on a certain metal layer.

Routing guides are created using the following function:

```
guideS :: Layer → Width → (Signal cell → Circ (Signal cell))
guideS l w sig = thinBlock (Guide pin l) w sig
  where
    pin = case sig of
        Signal nid (Constant _)     → (nid,0)
        Signal nid (Cell cp _ _ _) → (nid,cp)
```

This creates a thin block of width `w` and associates it with the passed-through signal. To guide arbitrary port structures, we define

```
guide :: PortStruct p (Signal cell) t ⇒ Layer → Width → (p → Circ p)
guide l w = mapPortM (guideS l w)
```

This will place a number of guides – all of the same width – side by side in the placement direction. Since `guide` is strictly more general than `guideS`, we never actually need to use `guideS`.

To view a picture of the floorplan together with the wires, as in figure 6.1, we use the following function:

[2] This definition uses *scoped type variables* to constrain the type of the parameter to `lengthFP`. Note that since `lengthFP` doesn't examine its parameter (appendix D), we can safely pass it an undefined value.

```
renderCircuitNets :: NodeId → String → Circ α → IO ()
renderCircuitNets nid title circ = renderFloorplan_ title fp wires
  where
    fp    = getFloorplan nid circ
    wires = routeNets fp
```

This uses the floorplan renderer that takes an additional list of lines as parameter. The list is created by `routeNets`,

```
routeNets :: Floorplan WiredBlock → [(Position,Position)]
```

which finds the set of guide positions for each wire and computes the minimal rectiliear spanning tree for each such guide cluster. For non-zero width guides, this function takes the guide position as being in the center of the block.

In figure 6.1, routing guides were already used to position the cell pins within the cell area (using `merged` placement), but this is hidden in the cell library, so the user doesn't have to deal with that.

Sklansky with guides

We will now see how refine `sklansky` from section 6.2 further by adding routing guides. There are basically two reasons why one may want to do that:

- Improving the visualization of the routing
- Helping the automatic router

Improved visualization is definitely useful when developing a layout. However, using guides to help the router is still an experimental feature, and although we believe it may be useful in certain cases, we have not been able to show this for any real designs.

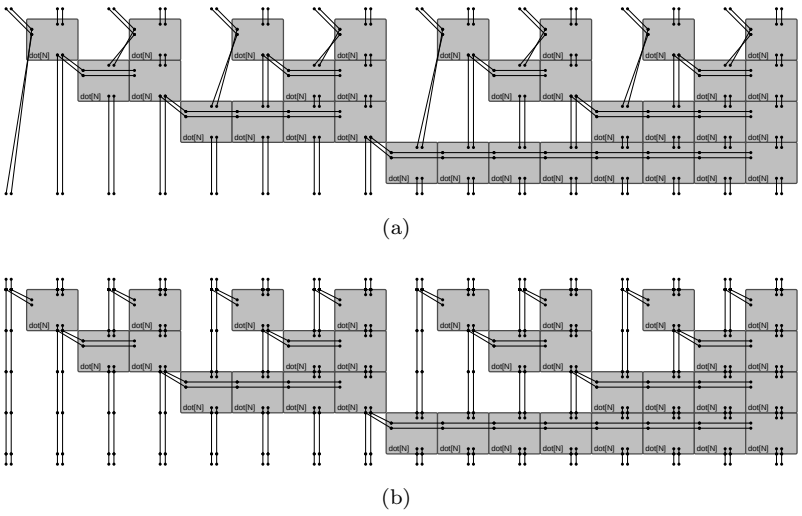
The first step in refining `sklansky` is nail down the positions of the input and output signals. The following helper function maps a guide on all elements of a list, and sprinkles space in between:

```
bus = rightwards 0 . mapM bus1
  where
    bus1 = space 2750 >=> guide 3 500 >=> space 1250
```

The sizes have been chosen to match the `dot` operator used in section 6.2. It happens to have a width of 5000, which we get by adding $2750 + 2 \cdot 500 + 1250$ ($2 \cdot 500$ because the guide gets a pair of signals). We can now wrap `sklansky` in a definition that applies `bus` on the inputs and outputs:

```
sklanskyIO op = downwards 0 .
    (bus >=> space 1000 >=> sklansky op >=> space 1000 >=> bus)
```

There is also some extra space inserted to make the picture more informative. Now it doesn't work to use `low` as input to `sklansky` anymore, since that would



(a)

(b)

Figure 9.1: Sklansky with routing guides

cause all input signals to be connected in one big net. Instead, we use `input` to make each input a unique signal:

```
*Main> let skl = sklanskyIO dot =<< replicateM 16 (input "in")
*Main> renderCircuitNets "skl" skl
```

The result is shown in figure 9.1(a), with the guides appearing as small blobs on the wires. In order to straighten up some of the lines, we may want to use guides inside the recursive calls as well. A new version of `sklansky`, which does this, is shown in figure 9.2, and the resulting layout in figure 9.1(b).

9.2 Timing analysis

In section 2.2.2, we modeled gate delay using an intrinsic delay constant T_{int} and an output resistance R_o. By multiplying R_o with the load capacitance (using Elmore's method when several stages were involved), we obtained the additional load-dependent delay for each output signal. We will refer to the total delay through a gate (intrinsic + load-dependent) as its *propagation delay*. So, in other words, propagation delay was given as a linear equation, with R_o as the slope and T_{int} as constant term.

A more general model (potentially more accurate) uses a custom non-linear function (often implemented as a lookup table [Cad02]) for each input/output path instead. In this chapter, we will implement such an analysis using the general interpretation framework from section 7.2.4. In reality, however, propa-

```
sklansky op [a] = space cellWidth [a]
sklansky op as  = downwards 1 $ do
    bus as
    let k = length as `div` 2
    let (ls,rs) = splitAt k as
    (ls',rs')   ← rightwards 0 $ liftM2 (,) (sklansky op ls)
                                            (sklansky op rs)
    rs'' ← rightwards 0 $ sequence [op (last ls', r) | r ← rs']
    bus (ls' ++ rs'')
```

Figure 9.2: `sklansky` with guides in each recursive call

gation delay is not completely determined by the load, it is also affected by the *transition times* of the involved input signals [Cad02]. For example, if an input changes value very slowly, this will make the propagation delay longer. Standard timing models therefore gain accuracy by also keeping track of transition times (distinguishing between rising and falling transitions). The actual Wired implementation includes transition delays, but in this section, we will ignore them in order to get a simpler presentation.

Figure 9.3 shows a timing interpretation for the previously defined library *Cells*. The values are delay-capacitance pairs, and delays are propagated forwards while capacitances flow backwards. Here we use simple linear equations with made-up parameters, but of course, they can easily be replaced by arbitrarily complex functions, including ones based on lookup tables. Note that we use lazy pattern matching (`~[...]`) in `prop`. This is generally needed with bidirectional analyses in order to avoid non-termination, since the underlying circular program relies on data being sufficiently lazy (section 7.1.1).

This interpretation can now be used to analyze Wired circuits (STA stands for Static Timing Analysis):

```
sta ::
    ( PortStruct α (Signal Cells) t
    , PortStruct τ Timing         t
    , PortStruct δ Delay          t
    ) ⇒ Circ α → δ

sta circ
    = mapPort timingDel
    $ interpret_ interpTiming [T 0 0, T 0 0] [] nid p
  where
    (p,nid) = runState (stripLayout circ) 2
```

Timing analysis seems to make most sense as a zero-arity analysis (i.e. we probably do not want to run the analysis with many different inputs, as one might with Boolean simulation). Therefore we use `interpret_`, which gives a more

```
type Delay       = Double
type Resistance  = Double
type Capacitance = Double

data Timing = T
       { timingDel :: Delay
       , timingCap :: Capacitance
       }

instance Port Timing Timing
  where
    port   = One
    unport = unOne

instance PortStruct Timing Timing ()

tInt = 5e-11 :: Delay
ro   = 100   :: Resistance
cg   = 1e-13 :: Capacitance

interpTiming :: Interpretation Cells Timing
interpTiming = Interp
    { defaultVal  = T 0 0
    , accumulator = λ(T d1 c1) (T d2 c2) → T (d1+d2) (c1+c2)
    , propagator  = prop
    }
  where
    load    = Just (T 0 cg)
    delay d = Just (T d 0)

    prop Inv ~[T din _, T _ cout] = [load, delay dout]
      where
        dout = din + tInt + ro*cout

    prop And2 ~[T din1 _, T din2 _, T _ cout] =
        [load, load, delay dout]
      where
        dout = max din1 din2 + tInt + ro*cout

    ...
```

Figure 9.3: Timing interpretation of the library *Cells*

low-level and flexible way of making interpretation functions than `interpret`.

Using suitable redefinitions of the gates from section 7.4 (using `liftGate`), we can now analyze the timing of the circuits from figure 7.5:

```
*Main> sta $ circ1 low
1.2e-10

*Main> sta $ circ2 (low,low)
1.1e-10

*Main> sta $ circ3 (low,low)
1.1e-10

*Main> sta $ circ4 low
(8.0e-11, 1.9e-10)
```

Since we used made-up parameter values, these values are not meaningful in any absolute sense. The parameters tell us that each gate has an intrinsic delay of 50ps and each fanout costs $100\Omega \cdot 0.1\text{pF} = 10\text{ps}$. For example, the first output of `circ4` has a fanout of three (section 7.2.4), and its delay is $50\text{ps} + 3 \cdot 10\text{ps} = 80\text{ps}$, as expected.

9.2.1 Wire-awareness

The simplest way to make the above analysis wire-aware is to just add in the estimated wire capacitance for each signal. The idea is simple: Use a function, similar to `routeNets` (section 9.1.4), to obtain the estimated length of each net:

```
estimateNetCaps :: Floorplan WiredBlock → [(Pin,Capacitance)]
```

Each net connects two or more pins, but there is always exactly one driver, so we can identify nets by the driver pins. The returned list associates a certain capacitance to each driver pin. Now we can take advantage of the extra `[(Pin,x)]` argument to `interpret_` (which we haven't used before) to add in the extra capacitances before the analysis:

```
staW circ
    = mapPort timingDel
    $ interpret_ interpTiming [T 0 0, T 0 0] extras 2 a
  where
    (a,fp) = fst $ runState (runLayoutT circ) 2
    extras = [ (p, T 0 c) | (p,c) ← estimateNetCaps fp]
```

This analysis completely ignores the resistance in wires, which means that it may greatly underestimate the delays of long wires. We are currently investigating how to combine an RC model for wires with a non-linear model for gates. We do not forsee any technical problems with this; it is purely because of time constraints on the project that this hasn't been done yet. However, as we will see in section 9.3.1, wire resistance is not a big source of inaccuracy in the examples we consider.

9.3 Export to CAD tools

Many CAD tools for physical design accept files in the Design Exchange Format (DEF) [Cad04]. This format has the right level of abstraction for Wired; it captures the netlist with optional cell placement (absolute) and optional "pre-routes" (routing guides) for the nets. By extracting a netlist from the Lava signal tree, and guide and cell positions from the floorplan (section 8.1.1), we can export our descriptions to DEF.

For example, the following simple circuit,

```
circ = rightwards 0 $ do
    (x1,x2) ← input "in"
    y1      ← and2    (x1,x2)
    (y2,y3) ← halfAdd (x1,y1)
    y4      ← and2    (y1,y2)
    return (y3,y4)
```

produces the DEF file in figure 9.4 (only the file body is shown). The file consists of three sections. First, the three components are listed. Each component has an instance name `CELLn` (where `n` happens to be the node *id* from the Lava signal tree), and an absolute position, obtained from the function `listAbsolute` from section 8.1.1. The next file section lists all nets in the circuit. Each net consists of a list of pins, where each pin refers to a certain signal name of a certain cell instance. The final section lists the primary inputs and outputs, and connects each input/output to a certain net. This example does not contain any routing guides.

We use the CAD tool Cadence Soc Encounter [Cad07], which allows us to read in a DEF file and do a complete automatic routing. Figure 9.5 shows the result of doing this for 16-bit Sklansky (compare to Figure 6.1(b)). In this case, the guides have been removed by the router.

Looking closer, we see from the diagonal line on each cell that every second row has been flipped. This is required by the design rules in order for the cells to share power rails. Such small tweaks are very easy to do in Wired; we simply replace the operator in each recursive call by a flipped version of itself:

```
sklansky op as  = downwards 1 $ do
    ...
    (ls',rs') ← rightwards 0 $ liftM2 (,) (sklansky (flipY . op) ls)
                                          (sklansky (flipY . op) rs)
    ...
```

9.3.1 Accuracy of timing analysis

For the realistic examples in this thesis, we use a 130nm cell library from ST-Microelectronics (`CORE9GPHS_HCMOS9_TEC_4.0`). The library comes with accu-

```
COMPONENTS 3 ;
- CELL4 AN2HSP + PLACED ( 0 0 ) N ;
- CELL5 HA1HSP + PLACED ( 2050 0 ) N ;
- CELL6 AN2HSP + PLACED ( 6560 0 ) N ;
END COMPONENTS

NETS 6 ;
- NET0 ( CELL5 S ) ( CELL6 B ) ;
- NET1 ( CELL5 CO ) ;
- NET2 ( CELL5 A ) ( CELL4 A ) ;
- NET3 ( CELL5 B ) ( CELL4 Z ) ( CELL6 A ) ;
- NET4 ( CELL4 B ) ;
- NET5 ( CELL6 Z ) ;
END NETS

PINS 4 ;
- PIN0_in + NET NET2 + DIRECTION INPUT ;
- PIN1_in + NET NET4 + DIRECTION INPUT ;
- PIN2 + NET NET1 + DIRECTION OUTPUT ;
- PIN3 + NET NET5 + DIRECTION OUTPUT ;
END PINS
```

Figure 9.4: DEF file procuced from `circ`

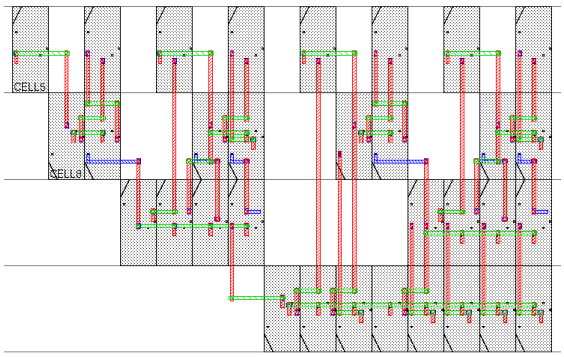


Figure 9.5: 16-bit Sklansky after auto routing in Soc Encounter

rate table-based models for propagation and transition delays. These could have been used directly in Wired, but in order to get a fast and simple analysis, we use linearized models instead. That is, each gate is modeled by two equations which approximate the corresponding table functions:

$$\begin{aligned} t_{prop} &= t_p + k_p \cdot t_{trans,in} + r_p \cdot C \\ t_{trans,out} &= t_t + k_t \cdot t_{trans,in} + r_t \cdot C \end{aligned}$$

Here, t_{prop} is the propagation delay, $t_{trans,in/out}$ is input/output transition time and C is the load capacitance.

We can get a feeling for the accuracy of our analysis by comparing it to the timing reported by Encounter. The following table shows the delays from analyzing Sklansky for the sizes $4 - 256$:

Size	*Encounter [ns]*	*Wired [ns]*	*Wired accuracy [%]*
4	0.176	0.198	12.5
8	0.299	0.319	6.7
16	0.458	0.466	1.7
32	0.677	0.664	−1.9
64	1.010	0.966	−4.4
128	1.571	1.473	−6.2
256	2.578	2.388	−7.4

This comparison is done using a two-input AND-gate as the operator. The different Sklansky instances have a wide range of different load capacitances and transition delays, so the comparison should be quite revealing. The overestimation for small sizes could be explained by the more approximate linear models used in Wired, and the increasing underestimation for larger sizes could be explained by the fact that Wired currently ignores wire resistance. On the other hand, this comparison also shows that wire resistance is not severely important for circuits of this size (in this slightly outdated technology).

9.4 Multiplier revisited

In section 4.2.3, we saw some details of a multiplier reduction tree in Hardwired. The conclusion from the example was that the top-level structure was reasonably simple, but the detailed definitions of the building blocks were way too complicated to be practically useful. We will now give the corresponding description in Wired, and see that it is very much simpler, without a significant loss of detail.

This time, we get away with a simpler block representation:

```
data Block = W | H | F
```

The matrix is generated like before:

```
*Main> redArrayBlock [1,2,3,4,5,6,5,4,3,2,1]
[[],[],[H],[H,F],[H,F,F],[H,F,F,F],[H,F,F,F],[W,F,F,F],[W,F,F],[W,F],[W]]
```

Figure 9.6 shows the function that builds the circuit from this representation.[3] The helper function `buildArray` constructs a row of columns using `rightwards` placement. Each column consists of a call to `buildColumn` with additional applications of `bus` to fix the positions of the inputs and outputs. In order to get a nicer interface, `redArray` just takes a list of signal lists as input and automatically figures out the parameters to `redArrayBlock`.

The individual building blocks are defined in figure 9.7. At this point, the advantage over the description in section 4.2.3 should be obvious. Here the entire block definition is simpler than the partial definition of *one* block in Hardwired. For example, `hBlock` now has two simple cases depending on whether or not it recieves a carry bit as input. Each case is just some data being threaded around a `halfAdd` component and a `bus` to guide the incoming signals.

The resulting circuit (for 12-bit multiplication) is shown in Figure 9.8.

9.5 Fast, low-power prefix networks

Sheeran has developed an algorithm for searching for parallel prefix networks with low power consumption and high speed [She07] (an offspring of the method in section 2.3). The algorithm returns a structural representation of the network and a polymorphic function which can be used to reconstruct the network with, for example, Lava gates as operators. Whenever the algorithm has to choose between different sub-circuits in a recursive decomposition, it does so by either picking the smallest delay (if the sub-network is on the critical path) or the smallest power (when not on the critical path).

The top-level code for the algorithm looks as follows:

```
prefix f p = memo pm
  where
    pm ([],_, w) = trywire ([],w)
    pm ([i],_,w) = trywire ([i],w)

    pm (is,_,w) | 2^h < length is = Fail

    pm (is,xs,w) = (bestOnE xs is f . dropFail)
        [ wrpC1 ds (prefix f p) (prefix p p)
          | ds ← topds1 g h m (length is)
        ]
```

The cost functions `f` and `p` are taken as parameters; `f` is supposed to be used for networks on the critical path and `p` for networks off the critical path. In the

[3] *Sig* is defined in section 7.2.1 as a synonym to *Signal Cells*.

```
bus = rightwards 0 . (space 500 >=> guide 1 400)

buildColumn
    :: [Block] → ([Maybe Sig], [Sig]) → Circ ([Sig], [Maybe Sig])
buildColumn [] (_,ps) = return (ps,[])
buildColumn (b:bs) (c:cs, ps) = do
    (ps',cs') ← buildColumn bs (cs,ps)
    unless (b==W) $ space 500 ()
    (ss,c')   ← circBlock b (c,ps')
    return (ss, c':cs')

buildArray :: Int → [[Block]] → [[Sig]] → Circ [[Sig]]
buildArray h bss pss = rightwards 0 $ build bss [] pss
  where
    build [] _ _ = return []

    build (bs:bss) cs (ps:pss) = do
        (ss,cs') ← downwards 0 $ do
            bus ps
            space (4500*h') ()
            (ss,cs') ← space 1000 =<< buildColumn
                (reverse bs) (cs ++ repeat Nothing, ps)
            bus ss
            return (ss,cs')
        sss ← build bss cs' pss
        return (ss:sss)
      where
        h' = h - length (filter (≠W) bs)

redArray :: [[Sig]] → Circ [[Sig]]
redArray pss = buildArray h bss (pss ++ repeat [])
  where
    bss = redArrayBlock $ map length pss
    h   = maximum $ map length bss
```

Figure 9.6: Definition of `redArray` which builds a Wired circuit from the matrix representation of reduction trees

```
type CircBlock = (Maybe Sig, [Sig]) → Circ ([Sig], Maybe Sig)

insert a bs = bs ++ [a]

hBlock :: CircBlock
hBlock (Nothing, ps@(p1:p2:ps')) = do
    bus ps
    (s,c) ← halfAdd (p1,p2)
    return (insert s ps', Just c)
hBlock (Just c, ps) = do
    bus ps
    let p1:p2:ps'' = insert c ps
    (s,c') ← halfAdd (p1,p2)
    return (insert s ps'', Just c')

fBlock :: CircBlock
fBlock (Nothing, ps@(p1:p2:p3:ps')) = do
    bus ps
    (s,c) ← fullAdd (p1,(p2,p3))
    return (insert s ps', Just c)
fBlock (Just c, ps) = do
    bus ps
    let p1:p2:p3:ps'' = insert c ps
    (s,c') ← fullAdd (p1,(p2,p3))
    return (insert s ps'', Just c')

wir :: CircBlock
wir (Just c, ps) = do
    ps' ← bus $ insert c ps
    return (ps', Nothing)

circBlock :: Block → CircBlock
circBlock W = wir
circBlock H = hBlock
circBlock F = fBlock
```

Figure 9.7: Building blocks of the reduction tree

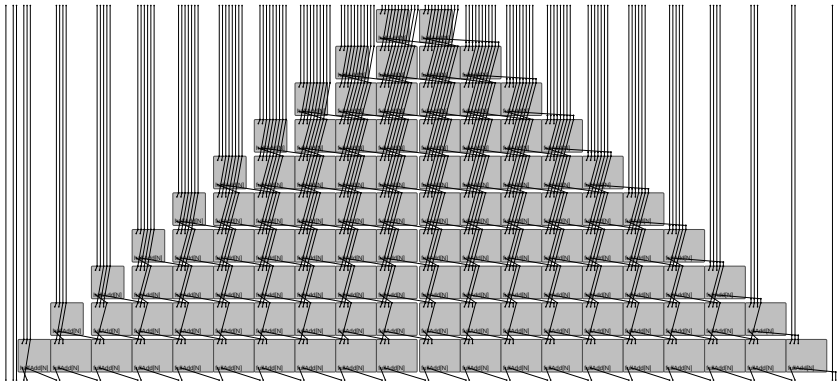


Figure 9.8: Multiplier reduction tree in Wired

recursive call, a list of candidates is created, and the cost function `f` is used to pick the best one. Each candidate has the form

```
wrpC1 ds (prefix f p) (prefix p p)
```

i.e. it is a composition of two smaller networks. The first one of these happens to be on the critical path, so it gets the same cost functions as the current call. The second network is not on the critical path, so instead it gets `p` as both cost functions so that `p` will be used in all selections inside that sub-network.

The performance models used in [She07] were crude: Delay was estimated as logical depth, and power as number of operators. However, since the presented algorithm abstracts away from the cost functions, it should be possible to improve the accuracy by just inserting more accurate functions.

The cost functions have the type `[Net] → Double`, where `[Net]` is the structural network representation. To make this presentation simpler, we will pretend that the `Circuit` type from section 4.1.1 is used instead of `[Net]`. In fact, the two representations are quite similar. The main difference is that `[Net]` contains some more information (not important for us), and it also leaves out some information which can be inferred by assuming that the network is a parallel prefix. Furthermore, the "coordinates" in the `[Net]` type are not necessarily meant to be geometrical positions – rather they represent bit position (significance) and logical depth – but it turns out that a geometrical interpretation of these numbers results in a reasonable network layout.

A redefinition of the `Circuit` type for Wired is given in figure 9.9, and figure 9.10 shows 8-bit Sklansky in this representation. In section 7.1.2, we defined a function for building a corresponding Lava circuit out of a `Circuit` representation. We will now use the same method, but add layout annotations to place the cells.

The helper function `declareSigs` is the same as before, but with the following

```
type Pos = (Int,Int)

data Comp = Comp
       { gate :: [Sig] → Circ Sig
       , inps :: [Pos]
       , out  :: Pos
       }

data Circuit = Circ
       { inputs  :: [Pos]
       , outputs :: [Pos]
       , comps   :: [Comp]
       }
```

Figure 9.9: Redefinition of *Circuit* for Wired

```
sklansky8 :: ((Sig,Sig) → Circ Sig) → Circuit
sklansky8 op = Circ
    { inputs  = zip [0..7] (repeat 0)
    , outputs = [(0,0),(1,1),(2,2),(3,2),(4,3),(5,3),(6,3),(7,3)]

    , comps =
        [ comp [(0,0),(1,0)] (1,1)
        , comp [(2,0),(3,0)] (3,1)
        , comp [(4,0),(5,0)] (5,1)
        , comp [(6,0),(7,0)] (7,1)
        , comp [(1,1),(2,0)] (2,2)
        , comp [(1,1),(3,1)] (3,2)
        , comp [(5,1),(6,0)] (6,2)
        , comp [(5,1),(7,1)] (7,2)
        , comp [(3,2),(4,0)] (4,3)
        , comp [(3,2),(5,1)] (5,3)
        , comp [(3,2),(6,2)] (6,3)
        , comp [(3,2),(7,2)] (7,3)
        ]
    }
  where
    comp = Comp (λ[a,b] → op (a,b))
```

Figure 9.10: 8-bit Sklansky in the *Circuit* representation

type signature:

```
declareSigs :: Circuit → LetT Sig Circ (Map Pos (Var Sig))
```

Since we do not want to assume anything about the order of components, it makes sense to use `merged` placement and just translate each cell to its position. Translation is done by the following function:

```
translate x y circ = leftwards 0 (trY >>= space (x*cellWidth))
  where
    trY = downwards 0 (circ >>= space (y*cellHeight))
```

The constants `cellWidth` and `cellHeight` should be the dimensions of the cell used in the network. Now we can define `buildCirc` as

```
buildCirc :: Circuit → ([Sig] → Circ [Sig])
buildCirc circ ss = merged 0 0 $ runLetT $ do
    posMap  ← declareSigs circ
    let pos = (posMap !)
    sequence_ [ pos i === s | (i,s) ← zip (inputs circ) ss ]
    mapM_ (buildComp pos) (comps circ)
    return (map (val.pos) (outputs circ))
  where
    buildComp pos (Comp gt is o@(x,y)) = do
        s ← lift $ translate x y $ gt (map (val.pos) is)
        pos o === s
```

The cost function for timing can now be defined as

```
staCirc :: Circuit → Delay
staCirc circ
    = maximum
    $ staW
    $ buildCirc circ
    $ replicate (length is) low
  where
    Circ is _ _ = circ
```

For power, we have just implemented a simple analysis on the structural representation – we do not even build a Wired circuit for it. The analysis is not a very accurate one; wire length estimation is quite crude (simply looking at horizontal distance between cells), and activity factors are estimated using simple assumptions. But the cell models are taken from the cell library datasheet and so should be accurate. The analysis is exposed as the function

```
powerCirc :: Circuit → Double
```

By plugging these two functions into the search

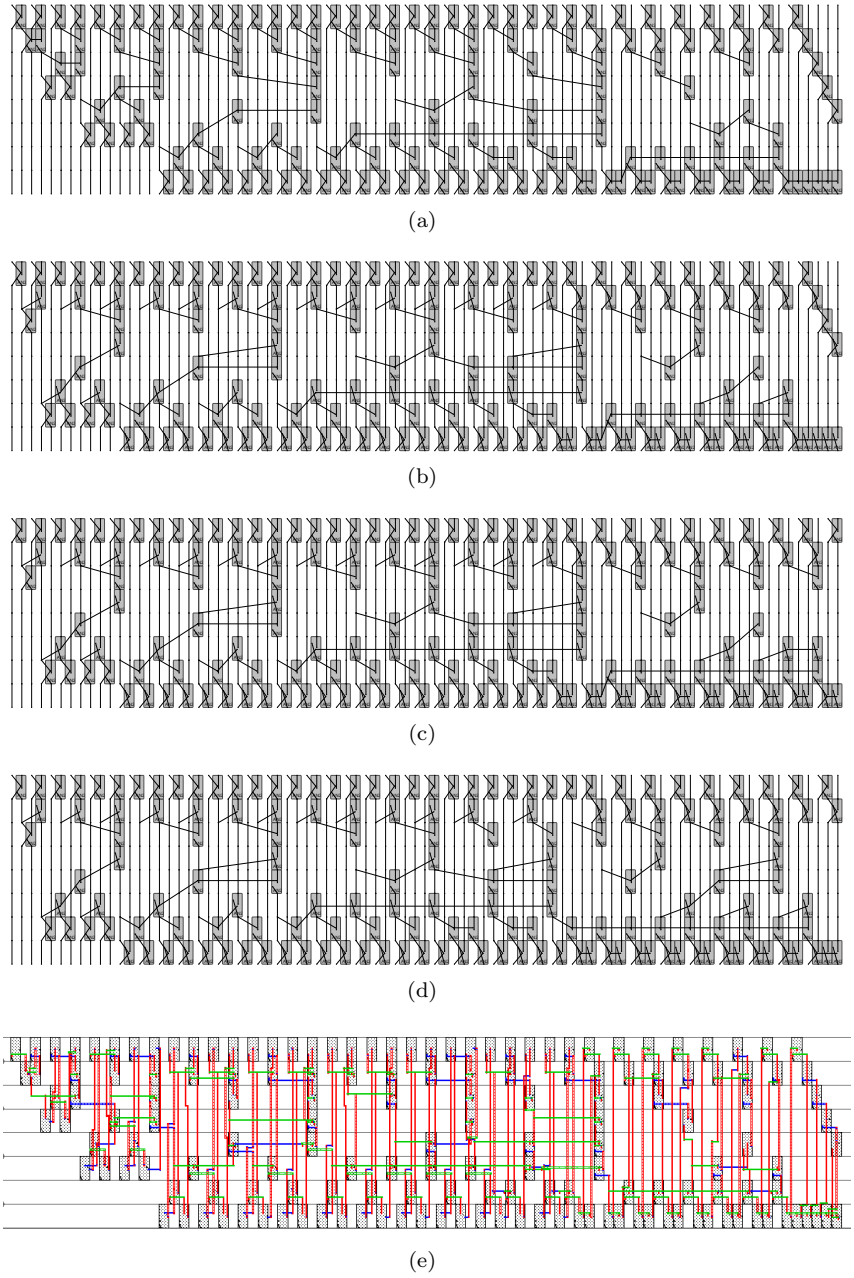
```
accuratePrefix = prefix staCirc powerCirc
```

we can now run the search algorithm with more accurate models, and thus possibly obtain better results. So far, we haven't found any remarkable results, but

we are currently experimenting with different options to see if there are situations where the more accurate algorithm pays off. To demonstrate the kind of design exploration we can do easily in Wired, we show the result of the parallel prefix search for width 85, logical depth 8 and varying maximum fanout (using a two-input AND-gate as operator):

Max fanout	*Delay [ns]*	*Power [mW]*
7	1.025	1.53
8	0.981	1.57
9	0.970	1.58
10	0.969	1.58

Figure 9.11 shows the resulting networks. It may seem counter-intuitive that higher fanout gives lower delay. This is because the search algorithm, according to common practice in theoretical treatment of parallel prefix, only counts the immediate fanout of each operator and ignores fanout that occurs on lower horizontal levels. Moreover, the algorithm only constrains the maximum fanout, while delay is determined by the *total fanout along the critical path*. It just happens, for this particular algorithm, that allowing higher fanout per level results in lower total fanout. Again, more experimentation is needed before we can draw any real conclusions about this search method.



(a)

(b)

(c)

(d)

(e)

Figure 9.11: Results of searching for optimal parallel prefix circuits with varying fanout: (a) $fanout = 7$, (b) $fanout = 8$, (c) $fanout = 9$, (d) $fanout = 10$, (e) $fanout = 7$, exported to Encounter

Part III

Epilogue

Chapter 10

Summary

10.1 Conclusions

I have presented the motivation behind this work: an urgent need in the IC industry for more adequate design methods. In particular, design flows that allow reasoning about the effects of routing wires are absolutely necessary for achieving efficient designs in modern chip technologies. The presented system, Wired, focuses on wire-aware design exploration for circuits that are designed at the placed netlist level, such as arithmetic units in high-performance microprocessors. This is still work in progress, but Wired seems to have very good potential to aid this kind of design, to reduce the design effort and to improve design quality.

Reaching a useful system was not easy. The first part of the thesis, which explores a very detailed circuit model with explicit wires, actually represents four out of the five years it took to complete my Ph.D. Thanks to the general logic programming library, LP, we managed to reach a working implementation of this system; however, after some experimenting with multiplier reduction trees, we decided that the model was too complicated and instead moved on to trying a much simpler model.

The second part of the thesis describes Wired as it is today. The circuit model uses a more abstract notion of wires, and this has resulted in a much simpler and more useful system. Although this part of the thesis is more interesting today, the new Wired system would not have been possible without the insights gained in the work on the first part. Wired is really the product of a chain of evolution steps, each of which was necessary for later improvements. In fact, the thesis also reflects my personal development – from a relatively inexperienced Haskell programmer with a poor background in computer science to a person who feels quite comfortable in the world of Haskell and its use as a host for

domain-specific languages.

Wired lives up to what was promised in the thesis title, "Functional Programming Enabling Flexible Hardware Design at Low Levels of Abstraction". The examples that have been presented demonstrate that Wired is indeed flexible and easy to use (with a certain degree of Haskell knowledge), and the link to CAD tools for physical design shows that the circuit model is faithful to the target technology. Furthermore, Wired gives the designer a handle on performance – wire effects included – which to our knowledge is unique for a system at this level of flexibility.

On the implementation side, Wired is interesting because of its simplicity and modularity. The following aspects of Haskell have played a very important role in this design: higher-order functions, monads, lazy evaluation and type classes.

Haskell is not directly suited for embedding the relational features of Wired – these would have been more naturally embedded in a functional-logic language like Curry. However, thanks to Haskell's support for monads, we were able to implement a general logic programming library in Haskell that offered exactly the features needed (with a slight syntactic overhead compared to Curry). This allowed us to stay within Haskell, which is a more mature and active language than Curry. Haskell is also the host of Lava, and it has always been our intention to have Wired and Lava as an integrated system (even if this was not the case in earlier implementations).

When implementing the simpler Wired system (with implicit wires), it became clear that we were only using the logic programming library in a quite restricted way, and we discovered that the same functionality could be achieved using circular programming techniques. This resulted in a very light-weight library – also monadic – for logical variables, in which most evaluation details are taken care of by Haskell's lazy semantics. While being more restricted than the general logic programming library, this new library also has the additional feature of allowing custom accumulation of constraints. This opened up for simple declarative descriptions of a range of different circuit analyses, including ones which have a bidirectional flow of information. The analyses demonstrated in this thesis were: Boolean simulation, fanout count and static timing analysis.

Layout combinators have been used in various systems to express relative placement of circuit blocks. In the second part of this thesis, we explored this style of description in a monadic setting, and this turned out to be very fruitful. It allows a clear separation between data flow and placement, and a kind of refinement, where the design is gradually transformed from a purely structural to a fully placed version. There are other systems which also have such a separation (for example Pebble [MLD02]), but the benefit here is that everything is done within Haskell. No special syntax is needed, and standard combinators for monadic programs can be reused.

The libraries for logical variables and layout are not necessarily restricted to their use in Wired; however, given their existence, the implementation of Wired

itself becomes quite simple. It can be explained as a combination of logical variables, Lava and layout. Apart from simplifying the implementation, this modularity should make it easier for new users to understand Wired simply by understanding its sub-systems.

10.2 Future work

Although Wired seems like a very promising approach to help the design of critical circuit components, we have not yet been able to show this for any realistic designs. In the future, we want to work on more realistic case-studies. For example, to make the parallel prefix search realistic, we have to incorporate gate sizing and buffer insertion into the search. A very interesting case-study once sizing has been worked out would be to run the algorithm for different technology nodes, say 130nm→90nm→65nm→45nm, and see what conclusions can be drawn from the resulting optimal structures. We also want to look at other kinds of circuits than the parallel prefix and reduction tree examples presented in this thesis.

We have seen that new cell libraries can be created quite easily in Wired, but it still requires quite a bit of typing for large libraries. We would like to (semi-)automate this process, so that complete cell libraries can be read in and produce the corresponding Haskell modules, including timing and power models. Wired currently has support for a few cells from a 130nm library. We would like to add support for the other libraries that we have access to (90nm and 65nm), as well as for the 45nm Open Cell Library from Nangate [Nan]. This last library could even be included in a publically available version of Wired.

We would like to improve the circuit analyses in Wired (without losing simplicity). Timing analysis should take wire resistance into account. Simple congestion estimation [TDNS07] could perhaps be used to gain a more realistic routing model. Finally, we also need a more general power analysis than the one used in this thesis.

Another interesting strand of work would be to look at custom floorplan transformations. For example, multiplier reduction trees can be laid out in different shapes [Eri03]. It may be that such variations are best described as post-placement transformations.

The layout library from chapter 8 is not restricted to standard cell designs (for example, see section 8.3). To broaden Wired's application domain, it would be very interesting to see if our system could be used, for example, for higher-level floorplanning of chip designs – especially since this is the level where wires start to get really problematic. Another possibility would be to also target FPGAs.

Wired seems to have many interesting possible applications waiting, and I can only hope that the work will continue after this thesis!

Bibliography

[ACS05] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*. Springer Verlag, October 2005.

[ADH+00] D. H. Allen, S. H. Dhong, H. P. Hofstee, J. Leenstra, K. J. Nowka, D. L. Stasiak, and D. F. Wendel. Custom circuit design as a driver of microprocessor performance. *IBM Journal of Research and Development*, 44(6):799–822, 2000.

[AGM+07] Cherif Andraos, Jennifer Gillenwater, Gregory Malecha, Angela Yun Zhu, Walid Taha, Jim Grundy, and John O'Leary. Synthesizable Verilog. In *HFL '07: International Workshop on Hardware Design and Functional Languages*, pages 139–153, 2007.

[AKL+07] Charles J. Alpert, Shrirang Karandikar, Zhuo Li, Gi-Joon Nam, Stephen Quay, Haoxing Ren, Cliff Sze, Paul G. Villarrubia, and Mehmet Yildiz. The nuts and bolts of physical synthesis. In *SLIP '07: Proceedings of the 2007 international workshop on System level interconnect prediction*, pages 89–94. ACM, 2007.

[ALES00] Atila Alvandpour, Per Larsson-Edefors, and Christer Svensson. GLMC: Interconnect length estimation by growth-limited multifold clustering. In *IEEE International Symposium on Circuits and Systems*, pages V–465 – V–468, 2000.

[BHK+87] B. Becker, G. Hotz, R. Kolla, P. Molitor, and H.-G. Osthof. Hierarchical design based on a calculus of nets. In *DAC '87: Proceedings of the 24th ACM/IEEE conference on Design automation*, pages 649–653. ACM Press, 1987.

[Bir84] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[BK82] Richard Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31(3):260–264, 1982.

[BKM01] Kenneth D. Boese, Andrew. B. Kahng, and Stefanus Mantik. On the relevance of wire load models. In *SLIP '01: Proceedings of the 2001 international workshop on System-level interconnect prediction*, pages 91–98. ACM Press, 2001.

[Bro03] Chappell Brown. SoC interconnect crisis: Path delays cancel speed increase. *EETimes: http://www.eetimes.com*, June 2003.

[Cad02] *Delay Calculation Algorithm Guide*. Product Version 5.0. Cadence Design Systems, 2002.

[Cad04] *LEF/DEF 5.5 Language Reference*. Product Version 5.5. Cadence Design Systems, 2004.

[Cad07] *Encounter User Guide*. Product Version 6.2. Cadence Design Systems, 2007.

[CKL+03] Yiu-Hing Chan, Prabhakar Kudva, Lisa Lacey, Greg Northrop, and Thomas Rosser. Physical synthesis methodology for high performance microprocessors. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 696–701. ACM Press, 2003.

[CL00] Koen Claessen and Peter Ljunglöf. Typed logical variables in Haskell. In *Proeedings of Haskell Workshop*. ACM SIGPLAN, 2000.

[Cla00] Koen Claessen. *An Embedded Language Approach to Hardware Description and Verification*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2000.

[Con01] Jason Cong. An interconnect-centric design flow for nanometer technologies. *Proceedings of the IEEE*, 89(4):505–528, Apr 2001.

[CRX03] Jason Cong, Michail Romesis, and Min Xie. Optimality and stability study of timing-driven placement algorithms. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, pages 472–478. IEEE Computer Society, 2003.

[CSS01] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *CHARME '01: Correct Hardware Design and Verification Methods*, volume 2144 of *LNCS*, pages 355–369. Springer-Verlag, 2001.

[Dre92] R. Drefenstadt. Parametric ASIC-design by CADIC. In *EDAC '92: Proceedings of the 3rd European Conference on Design Automation*, pages 267–271, Mar 1992.

[ELES+06] H. Eriksson, P. Larsson-Edefors, M. Sheeran, M. Sjalander, D. Johansson, and M. Scholin. Multiplier reduction tree with logarithmic logic depth and regular connectivity. In *ISCAS '06: IEEE International Symposium on Circuits and Systems, 2006*, pages 5–8. IEEE, May 2006.

[Eri03] Henrik Eriksson. *Efficient Implementation and Analysis of CMOS Arithmetic Circuits*. Ph.D. thesis, Department of Computer Engineering, Chalmers University of Technology, 2003.

[Eri08] Jack Erickson. What floorplan information is needed for synthesis? *EETimes: http://www.eetimes.com*, April 2008.

[Erk02] Levent Erkök. *Value Recursion in Monadic Computations*. Ph.D. dissertation, Oregon Graduate Institute School of Science Engineering, OHSU, 2002.

[FHWS00] Amir H. Farrahi, David J. Hathaway, M Wang, and Majid Sarrafzadeh. Quality of EDA CAD tools: Definitions, metrics and directions. In *ISQED '00: International Symposium on Quality in Electronic Design*, pages 395–495, 2000.

[FM97] M. Fujita and R. Murgai. Delay estimation and optimization of logic circuits: A survey. In *ASP-DAC '97: Proceedings of the Design Automation Conference*, pages 25–30, 1997.

[GL85] S.M. German and K.J. Lieberherr. Zeus: A language for expressing algorithms in hardware. *Computer*, 18(2):55–65, Feb 1985.

[GL95] Shaori Guo and Wayne Luk. Compiling Ruby into FPGAs. In *FPL '95: Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, pages 188–197. Springer, 1995.

[GL01] Shaori Guo and Wayne Luk. An integrated system for developing regular array designs. *Journal of Systems Architecture*, 47(3-4):315–337, 2001.

[GOPR02] Padmini Gopalakrishnan, Altan Odabasioglu, Lawrence Pileggi, and Salil Raje. An analysis of the wire-load model uncertainty problem. In *IEEE Transactions on Computer Aided Design of Circuits and Systems*, pages 23–31, 2002.

[GRSZ94] J. Griffith, G. Robins, J.S. Salowe, and Tongtong Zhang. Closing the gap: Near-optimal Steiner trees in polynomial time. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(11):1351–1365, Nov 1994.

[GTP97] R. Gupta, B. Tutuianu, and L.T. Pileggi. The Elmore delay as a bound for RC trees with generalized input signals. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(1):95–104, Jan 1997.

[HB92] Warren A. Hunt and Bishop C. Brock. A formal HDL and its use in the FM9001 verification. *Philosophical Transactions: Physical Sciences and Engineering*, 339(1652):35–47, Apr 1992.

[HE00] Zhijun Huang and Milos Ercegovac. Effect of wire delay on the design of prefix adders in deep-submicron technology. In *34th Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1713–1717. IEEE Computer Society Press, 2000.

[Her06] Christoph Herrmann. *Type-Sensitive Size Parameterization of Circuit Designs by Metaprogramming*. Number MIP-0601. Fakultät für Mathematik und Informatik, Universität Passau, 2006.

[HF92] Paul Hudak and Joseph H. Fasel. *A gentle introduction to Haskell*, volume 27(5). ACM SIGPLAN Notices, 1992.

[HHM99] Ron Ho, Mark A. Horowitz, and Kenneth W. Mai. The future of wires. *Invited Workshop Paper for SRC Conference*, 1999.

[Hin00] Ralf Hinze. Deriving backtracking monad transformers. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 186–197. ACM Press, 2000.

[HJSB80] Jack Holloway, Guy Lewis Steele Jr., Gerald Jay Sussman, and Alan Bell. *The SCHEME-79 Chip*. Number Technical report 599. Artificial Intelligence Laboratory, MIT, 1980.

[HKMN95] Michael Hanus, Herbert Kuchen, and Jose Moreno-Navarro. Curry: A truly functional logic language. In *ILPS '95: Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[Ho03] Ron Ho. *On-Chip Wires: Scaling and Efficiency*. Ph.D. thesis, Stanford University, 2003.

[Hoo] Hoogλe: The Haskell API Search Engine. *http://www.haskell.org/hoogle/*.

[HR05] Warren A. Hunt and Erik Reeber. Formalization of the DE2 language. In *CHARME '05: Correct Hardware Design and Verification Methods*, volume 3725 of *LNCS*, pages 20–34. Springer, Oct 2005.

[HRW92] Frank K. Hwang, Dana S. Richards, and Pawel Winter. *The Steiner Tree Problem*. Amsterdam: North Holland, 1992.

[Hud98] P. Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[Hut90] Graham Hutton. Functional programming with relations. In *Proc. Third Glasgow Workshop on Functional Programming, Workshops in Computing*, pages 126–140. Springer-Verlag, 1990.

[ITR05] Semiconductor Industry Association. The International Technology Roadmap for Semiconductors, 2005 edition. International SEMATECH: Austin, TX. 2005.

[J+03] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[JN94] Neil D. Jones and Flemming Nielsen. Abstract interpretation: A semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, pages 527–636. OUP, 1994.

[Joh84] Steven D. Johnson. Applicative programming and digital design. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 218–227. ACM, 1984.

[Joh98] Thomas Johnsson. Efficient graph algorithms using lazy monolithic arrays. *Journal of Functional Programming*, 8(4):323–333, 1998.

[Jon92] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[Jon95] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136. Springer-Verlag, 1995.

[JS90] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In *Formal Methods for VLSI Design*. North-Holland, 1990.

[JS93] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Mathematics of Program Construction*, volume 669 of *LNCS*, pages 208–232. Springer-Verlag, 1993.

[LHS] lhs2TeX version 1.13. *http://people.cs.uu.nl/andres/lhs2tex/*.

[LM98] Wayne Luk and Steve McKeever. Pebble: A language for parametrised and reconfigurable hardware design. In *Field-Programmable Logic and Applications*, volume 1482 of *LNCS*, pages 9–18. Springer-Verlag, 1998.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *PLAN '99: Proceedings of the 2nd conference on domain-specific languages*, pages 109–122. ACM Press, 1999.

[LM03] J.R. Lewis and B. Martin. Cryptol: High assurance, retargetable crypto development and validation. In *MILCOM 2003: Military Communications Conference*, volume 2, pages 820–825. IEEE, Oct 2003.

[LNS+82] Richard J. Lipton, Stephen C. North, Robert Sedgewick, Jacobo Valdes, and Gopalakrishanan Vijayan. ALI: A procedural language

to describe VLSI layouts. In *DAC '82: Proceedings of the 19th conference on Design automation*, pages 467–474. IEEE Press, 1982.

[Luk90] Wayne Luk. Analysing parametrised designs by non-standard interpretation. In *International Conference on Application-Specific Array Processors*, pages 133–144. IEEE Computer Society Press, 1990.

[LW01] Minghorng Lai and D. F. Wong. Slicing tree is a complete floorplan representation. In *DATE '01: Design, Automation and Test in Europe*, pages 228–232, 2001.

[Mar08] Andrew K. Martin. Bridging the gap between abstract RTL and bit-level designs. In *DCC '08: Int. Workshop on Designing Correct Circuits*, page 72, March 2008.

[McG06] Dylan McGrath. AMD researcher calls for design regularity. *EETimes: http://www.eetimes.com*, Feb 2006.

[MKWS04] Nir Magen, Avinoam Kolodny, Uri Weiser, and Nachum Shamir. Interconnect-power dissipation in a microprocessor. In *SLIP '04: Proceedings of the 2004 international workshop on System level interconnect prediction*, pages 7–13. ACM, 2004.

[MLD02] Steve McKeever, Wayne Luk, and Arran Derbyshire. Compiling hardware descriptions with relative placement information for parametrised libraries. In *FMCAD '02: Formal Methods in Computer-Aided Design*, LNCS, pages 342–359. Springer-Verlag, 2002.

[MS03] Alan Mycroft and Richard Sharp. Higher-level techniques for hardware description and synthesis. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(3):271–297, May 2003.

[Nan] Nangate Inc. *http://www.nangate.com*.

[NAR07] Matthew Naylor, Emil Axelsson, and Colin Runciman. A functional-logic library for Wired. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 37–48. ACM, 2007.

[Nik04] R. Nikhil. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *MEMOCODE '04: International Conference on Formal Methods and Models for Co-Design*, June 2004.

[NTR97] Semiconductor Industry Association. National Technology Roadmap for Semiconductors. 1997.

[O'D95] John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *FPLE '95: Symposium on Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 195–214. Springer-Verlag, 1995.

[O'D04] John O'Donnell. Embedding a hardware description language in template haskell. In *Domain-Specific Program Generation*, volume 3016/2004 of *LNCS*, pages 143–164. Springer, Nov 2004.

[OVL96] Vojin G. Oklobdzija, David Villeger, and Simon S. Liu. A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *IEEE Trans. Computers*, 45(3):294–306, 1996.

[PAB+06] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation Cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, Jan 2006.

[PL05] Oliver Pell and Wayne Luk. Quartz: A framework for correct and efficient reconfigurable design. In *ReConFig '05: International Conference on Reconfigurable Computing and FPGAs*. IEEE Computer Society Press, 2005.

[PMB08] Stephen M. Plaza, Igor L. Markov, and Valeria Bertacco. Optimizing non-monotonic interconnect using functional simulation and logic restructuring. In *ISPD '08: Proceedings of the 2008 international symposium on Physical design*, pages 95–102. ACM, 2008.

[PT08] Gordon J. Pace and Christian Tabone. Access to circuit generators in embedded HDLs. In *DCC '08: Int. Workshop on Designing Correct Circuits*, pages 74–85, March 2008.

[Rab03] Jan M. Rabaey. *Digital Integrated Circuits*. Prentice Hall, 2003.

[RKG+92] C. Ramachandran, F. J. Kurdahi, D. D. Gajski, A. C.-H. Wu, and V. Chaiyakul. Accurate layout area and delay modeling for system level design. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 355–361. IEEE Computer Society Press, 1992.

[RPH83] J. Rubinstein, P. Penfield, and M.A. Horowitz. Signal delay in RC tree networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(3):202–211, July 1983.

[Seg06] Carl Seger. The design of a floating point execution unit using the Integrated Design and Verification (IDV) system. In *DCC '06: Int. Workshop on Designing Correct Circuits*, March 2006.

[She83] Mary Sheeran. *μFP, an algebraic VLSI design language*. D. Phil. thesis, Oxford University, 1983.

[She03] Mary Sheeran. Finding regularity: Describing and analysing circuits that are almost regular. In *CHARME '03: Correct Hardware Design and Verification Methods*, volume 2860 of *LNCS*, pages 4–18. Springer-Verlag, 2003.

[She04] Mary Sheeran. Generating fast multipliers using clever circuits. In *FMCAD '04: Formal Methods in Computer-Aided Design*, volume 3312 of *LNCS*, pages 6–20. Springer-Verlag, 2004.

[She07] Mary Sheeran. Parallel prefix network generation: an application of functional programming. In *HFL '07: International Workshop on Hardware Design and Functional Languages*, pages 21–46, 2007.

[Sin92] Satnam Singh. Circuit analysis by non-standard interpretation. In *Designing Correct Circuits*, volume A-5 of *IFIP Transactions*, pages 119–138. North-Holland, 1992.

[Sin00] Satnam Singh. Death of the RLOC? In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 145–152. IEEE Computer Society Press, 2000.

[SK98] Dennis Sylvester and Kurt Keutzer. Getting to the bottom of deep submicron. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 203–211. ACM Press, 1998.

[Skl60] J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(6):226–231, 1960.

[SMOR98] Paul F. Stelling, Charles U. Martel, Vojin G. Oklobdzija, and R. Ravi. Optimal circuits for parallel multipliers. *IEEE Trans. Computers*, 47(3):273–285, 1998.

[SN00] Lou Scheffer and Eric Nequist. Why interconnect prediction doesn't work. In *SLIP '00: Proceedings of the 2000 international workshop on System-level interconnect prediction*, pages 139–144. ACM Press, 2000.

[Spo02] Joel Spolsky. The law of leaky abstractions. *http://www.joelonsoftware.com/articles/LeakyAbstractions.html*, Nov 2002.

[SS99] Michael Spivey and Silvija Seres. Embedding Prolog in Haskell. In *Proeedings of Haskell Workshop*, 1999.

[TDNS07] Taraneh Taghavi, Foad Dabiri, Ani Nahapetian, and Majid Sarrafzadeh. Tutorial on congestion prediction. In *SLIP '07: Proceedings of the 2007 international workshop on System level interconnect prediction*, pages 15–24. ACM, 2007.

[TKP+04] Louise Trevillyan, David Kung, Ruchir Puri, Lakshmi N. Reddy, and Michael A. Kazda. An integrated environment for technology closure of deep-submicron IC designs. *IEEE Design&Test of Computers*, pages 14–22, 2004.

[UK02] Junhyung Um and Taewhan Kim. Layout-aware synthesis of arithmetic circuits. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 207–212. ACM Press, 2002.

[Wad93] Philip Wadler. Monads for functional programming. In *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

[Zim98] Reto Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*. Ph.D. thesis, Swiss Federal Institute of Technology (ETH) Zurich, Hartung-Gorre Verlag, 1998.

Appendix A

Terminology

This is a quick summary of some overloaded words that are used in the thesis. Cross-references are underlined.

Circuit:
Means any kind of electronic network at any level of abstraction. In the VLSI field, "circuit" more commonly stands for a transistor-level schematic, but that is not the main meaning in this thesis.

Combinator:
Generally refers to a (higher-order) function for building objects in an embedded domain-specific language. In this thesis it most often refers to the special case of combining sub-circuits into bigger ones.

Connection pattern:
In general, denotes a combinator for circuits. There is some tendency in this text to use the word for *recursive* (regular) combinators, but this is not always the case.

Design:
A non-trivial circuit.

Floorplan:
The hierarchical layout of sub-modules in a circuit. In the system presented in this thesis, the floorplan hierarchy often goes all the way down to the cells, which means that the floorplan implicitly determines the placement.

Layout:
In the VLSI field, "layout" usually means the polygons that eventually make up the transistors and wires on the chip. In this thesis we use it in the more general sense: "how things are laid out".

Placement:
The layout of cells in a standard cell design.

Appendix B

Knot

```
module Knot where

import Data.Map (Map,fromList,fromListWith,(!),findWithDefault)
import Control.Monad.Reader
import Control.Monad.Writer

newtype Knot i x α = Knot (ReaderT (Map i x) (Writer [(i,x)]) α)
        deriving
          ( Monad
          , MonadReader (Map i x)
          , MonadWriter [(i,x)]
          )

newtype KnotT i x m α =
          KnotT (ReaderT (Map i x) (WriterT [(i,x)] m) α)
        deriving
          ( Monad
          , MonadReader (Map i x)
          , MonadWriter [(i,x)]
          )

instance MonadTrans (KnotT i x)
  where
    lift = KnotT . lift . lift

class (Ord i, MonadReader (Map i x) m, MonadWriter [(i,x)] m) ⇒
      MonadKnot i x m | m → i x
  where
    askKnot    :: i → m x
    askKnotDef :: x → i → m x
    (*=)       :: i → x → m ()
```

```
instance (Ord i, MonadReader (Map i x) m, MonadWriter [(i,x)] m) ⇒
         MonadKnot i x m
  where
    askKnot i        = asks (! i)
    askKnotDef def i = asks (findWithDefault def i)
    i *= x           = tell [(i,x)]

accKnot :: Ord i ⇒ (x → x → x) → Knot i x α → (α, Map i x)
accKnot acc (Knot knot) = (a,mapping)
  where
    (a,ass) = runWriter $ runReaderT knot mapping
    mapping = fromListWith acc ass

tieKnot :: Ord i ⇒ Knot i x α → (α, Map i x)
tieKnot = accKnot (error "tieKnot: Over-constrained")


accKnotT
    :: (Ord i, MonadFix m)
    ⇒ (x → x → x) → KnotT i x m α → m (α, Map i x)

accKnotT acc (KnotT knot) = mdo
    (a,ass) ← runWriterT $ runReaderT knot mapping
    let mapping = fromListWith acc ass
    return (a,mapping)


tieKnotT :: (Ord i, MonadFix m) ⇒ KnotT i x m α → m (α, Map i x)
tieKnotT = accKnotT (error "tieKnot: Over-constrained")
```

Appendix C

Let

```
module Let where

import Data.Map
import Control.Monad.Reader
import Control.Monad.Writer
import Control.Monad.State

import Knot

type VarId = Int

newtype Let x α = Let (StateT VarId (Knot VarId x) α)
        deriving
          ( Monad
          , MonadReader (Map VarId x)
          , MonadWriter [(VarId,x)]
          , MonadState VarId
          )

newtype LetT x m α = LetT (StateT VarId (KnotT VarId x m) α)
        deriving
          ( Monad
          , MonadReader (Map VarId x)
          , MonadWriter [(VarId,x)]
          , MonadState VarId
          )

data Var x = Var VarId x

instance MonadTrans (LetT x)
  where
    lift = LetT . lift . lift
```

```
class (MonadState VarId m, MonadKnot VarId x m) ⇒
      MonadLet x m | m → x
  where
    free :: m (Var x)

instance (MonadState VarId m, MonadKnot VarId x m) ⇒ MonadLet x m
  where
    free = do
      vid ← get
      put (succ vid)
      x ← askKnot vid
      return (Var vid x)

infix 1 ===

(===) :: MonadLet x m ⇒ Var x → x → m ()
Var vid _ === x = vid *= x

val :: Var x → x
val (Var _ x) = x

runLet :: Let x α → α
runLet (Let ma) = fst $ fst $ tieKnot $ runStateT ma 0

runLetT :: MonadFix m ⇒ LetT x m α → m α
runLetT (LetT ma) = liftM (fst.fst) $ tieKnotT $ runStateT ma 0
```

Appendix D

WiredLava

```
module WiredLava where

import qualified Data.List as List
import Data.Foldable (Foldable)
import qualified Data.Foldable as Fold
import Data.Traversable (Traversable, traverse)
import qualified Data.Traversable as Trav
import Control.Applicative
import Control.Monad.State

import Knot

type Name    = String
type CellPin = Int
type NumOuts = Int
type NodeId  = Int
type Pin     = (NodeId,CellPin)

data Node cell s
       = Constant Name
       | Cell CellPin cell NumOuts [s]

data Signal cell = Signal NodeId (Node cell (Signal cell))

instance Functor (Node cell)
  where
    fmap f (Constant nm)        = Constant nm
    fmap f (Cell cp cid n inps) = Cell cp cid n (map f inps)

instance Foldable (Node cell)
  where
    foldr f x (Constant _)      = x
    foldr f x (Cell _ _ _ inps) = List.foldr f x inps
```

```
instance Traversable (Node cell)
  where
    traverse f (Constant nm) = pure (Constant nm)
    traverse f (Cell cp cid n inps)
        = pure (Cell cp cid n) <*> traverse f inps

type Lava = State NodeId

data PortTree s
       = One  {unOne :: s}
       | List [PortTree s]

instance Functor PortTree
  where
    fmap f (One s)   = One (f s)
    fmap f (List ps) = List $ map (fmap f) ps

instance Foldable PortTree
  where
    foldr f x (One s)   = f s x
    foldr f x (List ps) = List.foldr (flip $ Fold.foldr f) x ps

instance Traversable PortTree
  where
    traverse f (One s)   = pure One  <*> f s
    traverse f (List ps) = pure List <*> traverse (traverse f) ps

class Port p s | p → s
  where
    port   :: p → PortTree s
    unport :: PortTree s → p

instance Port (Signal cell) (Signal cell)
  where
    port   = One
    unport = unOne

instance Port () ()
  where
    port   = One
    unport = unOne

instance Port Bool Bool
  where
    port   = One
    unport = unOne

instance Port Int Int
  where
    port   = One
    unport = unOne
```

```
instance Port Double Double
  where
    port   = One
    unport = unOne

instance Port p s ⇒ Port [p] s
  where
    port             = List . map port
    unport (List ps) = map unport ps

instance (Port p1 s, Port p2 s) ⇒ Port (p1,p2) s
  where
    port (p1,p2)          = List [port p1, port p2]
    unport (List [p1,p2]) = (unport p1, unport p2)

class Port p s ⇒ PortStruct p s t | p → s t, s t → p

instance PortStruct (Signal cell) (Signal cell) ()
instance PortStruct ()            ()            ()
instance PortStruct Bool          Bool          ()
instance PortStruct Int           Int           ()
instance PortStruct Double        Double        ()

instance PortStruct p s t ⇒ PortStruct [p] s [t]

instance (PortStruct p1 s t1, PortStruct p2 s t2) ⇒
         PortStruct (p1,p2) s (t1,t2)

mapPort
    :: (PortStruct pa sa t, PortStruct pb sb t)
    ⇒ (sa → sb) → (pa → pb)
mapPort f = unport . fmap f . port

mapPortM
    :: (PortStruct pa sa t, PortStruct pb sb t, Monad m)
    ⇒ (sa → m sb) → (pa → m pb)
mapPortM f = liftM unport . Trav.mapM f . port

class Port p s ⇒ PortFixed p s | p → s
  where
    lengthFP   :: p → Int
    fromListFP :: [s] → p

instance PortFixed (Signal cell) (Signal cell)
  where
    lengthFP       = const 1
    fromListFP [s] = s
```

```
instance (PortFixed p1 s, PortFixed p2 s) ⇒ PortFixed (p1,p2) s
  where
    lengthFP ~(p1,p2)
        = lengthFP (undefined `asTypeOf` p1)
        + lengthFP (undefined `asTypeOf` p2)

    fromListFP ss = (fromListFP ss1, fromListFP ss2)
      where
        (ss1,ss2) = splitAt (lengthFP (undefined :: p1)) ss

constant :: NodeId → Name → Signal cell
constant nid nm = Signal nid (Constant nm)

cellList :: NumOuts → cell → [Signal cell] → Lava [Signal cell]
cellList n cid ins = do
    nid ← get
    put (succ nid)
    return [Signal nid $ Cell cp cid n ins | cp ← [0 .. n-1]]

cell
    :: ∀ pi po cell l
     . (Port pi (Signal cell), PortFixed po (Signal cell))
    ⇒ cell → pi → Lava po

cell cid pi = liftM fromListFP $ cellList n cid ins
  where
    n   = lengthFP (undefined :: po)
    ins = Fold.toList $ port pi

sigPin :: Signal cell → Pin
sigPin (Signal nid comp) = (nid,cp)
  where
    cp = case comp of
        Cell cp _ _ _ → cp
        _             → 0
```

Appendix E

Interpretation

```
module Interpretation where

import Control.Monad.State
import qualified Data.Foldable as Fold

import Knot
import WiredLava
import Traversal

data Interpretation cell x = Interp
       { constants   :: [x]
       , defaultVal  :: x
       , accumulator :: x → x → x
       , propagator  :: cell → ([x] → [Maybe x])
       }

interpretSig :: Interpretation cell x → Signal cell → Traversal s x x
interpretSig interp sig =
    interpretS sig >> askKnotDef (defaultVal interp) (sigPin sig)
  where
    interpretS (Signal nid ~(Cell _ cid n ins)) = do
      visited ← isVisited nid
      setDone nid
      unless visited $ do
          mapM_ interpretS ins
          let pins = map sigPin ins ++ [(nid,cp) | cp ← [0 .. n-1]]
          vals ← mapM (askKnotDef $ defaultVal interp) pins
          let vals' = propagator interp cid vals
          sequence_ [ pin *= x | (pin, Just x) ← zip pins vals' ]

interpretConstants :: [x] → Traversal s x ()
interpretConstants cs = sequence_
    [ (nid,0)*=c >> setDone nid | (nid,c) ← zip [0..] cs ]
```

```
interpret_
    :: (PortStruct ps (Signal cell) t, PortStruct px x t)
    ⇒ Interpretation cell x → [x] → [(Pin,x)] → NodeId → ps → px

interpret_ interp cs es nid ps = fst $
    runTraversal (0,nid) (accumulator interp)
      ( do interpretConstants cs
           mapM_ (uncurry (*=)) es
           mapPortM (interpretSig interp) ps
      )


inputToSig :: (PortStruct px x t, PortStruct ps (Signal cell) t) ⇒
    NodeId → px → ps
inputToSig nid = flip evalState nid . mapPortM toSig
  where
    toSig x = do
      nid ← get
      put (succ nid)
      return $ Signal nid $ Constant ("input" ++ show nid)


interpret
    :: ( PortStruct pxi x             ti
       , PortStruct psi (Signal cell) ti
       , PortStruct pso (Signal cell) to
       , PortStruct pxo x             to
       )
    ⇒ Interpretation cell x
     → (psi → Lava pso)
     → (pxi → pxo)

interpret interp fs pxi = interpret_ interp cs [] nid ps
  where
    psi      = inputToSig (length $ constants interp) pxi
    cs       = constants interp ++ Fold.toList (port pxi)
    (ps,nid) = runState (fs psi) (length cs)
```

Appendix F

Traversal

```
module Traversal where

import Control.Monad.Reader
import Control.Monad.ST
import Control.Monad.Writer
import Data.Array.ST
import Data.Map (Map)

import Knot
import WiredLava

data Status = NotVisited | Done

type Visit s = ReaderT (STArray s NodeId Status) (ST s)

newtype Traversal s x a =
          Traversal { unTraversal :: KnotT Pin x (Visit s) a }
    deriving (Monad, MonadReader (Map Pin x), MonadWriter [(Pin,x)])

runTraversal
    :: (NodeId,NodeId)
     → (x → x → x)
     → (∀ s . Traversal s x a) → (a, Map Pin x)

runTraversal bnds acc trav = runST st
  where
    st  = newArray bnds NotVisited
      >>= runReaderT (accKnotT acc (unTraversal trav))

nodeStatus :: NodeId → Traversal s x Status
nodeStatus i = Traversal $ lift $ (lift . flip readArray i) =<< ask
```

```
setNodeStatus :: NodeId → Status → Traversal s x ()
setNodeStatus i stat = Traversal $ lift $ do
    stats ← ask
    lift $ writeArray stats i stat


setDone :: NodeId → Traversal s x ()
setDone i = setNodeStatus i Done

isVisited :: NodeId → Traversal s x Bool
isVisited i = do
    st ← nodeStatus i
    return $ case st of
      NotVisited → False
      _          → True
```

Appendix G

Floorplan

```
module Floorplan where

import Control.Monad.Writer

import WiredLava (Name)

type Alignment = Rational
type Width     = Int
type Height    = Int

data Placement
       = Unspecified
       | Merged Alignment {- x -} Alignment {- y -}
       | Row Direction Alignment

data Direction = Rightwards | Leftwards | Upwards | Downwards

data Floorplan bl
       = Block Width Height bl
       | Comb Placement [Floorplan bl]

type BlockType = Maybe Name

class IsBlock bl
  where
    spaceBlock :: bl
    toBlock    :: bl → BlockType

instance IsBlock BlockType
  where
    spaceBlock = Nothing
    toBlock    = id
```

```
class Transformable a
  where
    flipX   :: a → a
    flipY   :: a → a
    rotate_ :: Int → a → a

instance Transformable BlockType
  where
    flipX   = id
    flipY   = id
    rotate_ = const id

instance Transformable Direction
  where
    flipX Rightwards = Leftwards
    flipX Leftwards  = Rightwards
    flipX dir        = dir

    flipY Upwards   = Downwards
    flipY Downwards = Upwards
    flipY dir       = dir

    rotate_ n dir = iterate rot dir !! n
      where
        rot Rightwards = Upwards
        rot Leftwards  = Downwards
        rot Upwards    = Leftwards
        rot Downwards  = Rightwards

instance Transformable Placement
  where
    flipX (Merged alx aly)    = Merged (1-alx) aly
    flipX (Row Rightwards al) = Row Leftwards  al
    flipX (Row Leftwards  al) = Row Rightwards al
    flipX (Row up_down    al) = Row up_down    (1-al)
    flipX pl                  = pl

    flipY (Merged alx aly)    = Merged alx (1-aly)
    flipY (Row Upwards    al) = Row Downwards  al
    flipY (Row Downwards  al) = Row Upwards    al
    flipY (Row left_right al) = Row left_right (1-al)
    flipY pl                  = pl

    rotate_ n pl = iterate rot pl !! n
      where
        rot (Merged alx aly)    = Merged (1-aly) alx
        rot (Row Rightwards al) = Row Upwards    (1-al)
        rot (Row Leftwards  al) = Row Downwards  (1-al)
        rot (Row Upwards    al) = Row Leftwards  al
        rot (Row Downwards  al) = Row Rightwards al
        rot pl                  = pl
```

```
instance Transformable bl ⇒ Transformable (Floorplan bl)
  where

    flipX (Block w h bl) = Block w h (flipX bl)
    flipX (Comb pl fps)  = Comb (flipX pl) (map flipX fps)

    flipY (Block w h bl) = Block w h (flipY bl)
    flipY (Comb pl fps)  = Comb (flipY pl) (map flipY fps)

    rotate_ n (Block x y bl)
      | even n    = Block x y (rotate_ n bl)
      | otherwise = Block y x (rotate_ n bl)

    rotate_ n (Comb pl fps) = Comb (rotate_ n pl) $ map (rotate_ n) fps


rotate :: Transformable a ⇒ Int → a → a
rotate n = rotate_ ((n`mod`4 + 4) `mod` 4)
```

Appendix H

Layout

```
module Layout where

import Control.Monad.Reader
import Control.Monad.Writer

import Floorplan

newtype Layout bl α = Layout
          (ReaderT Placement (Writer [Floorplan bl]) α)
        deriving
          ( Monad
          , MonadReader Placement
          , MonadWriter [Floorplan bl]
          , MonadFix
          )

newtype LayoutT bl m α = LayoutT
          (ReaderT Placement (WriterT [Floorplan bl] m) α)
        deriving
          ( Monad
          , MonadReader Placement
          , MonadWriter [Floorplan bl]
          , MonadFix
          )

runLayout :: Layout bl α → (α, Floorplan bl)
runLayout (Layout m) = (a, Comb Unspecified fps)
  where
    (a,fps) = runWriter $ flip runReaderT Unspecified m

runLayoutT :: Monad m ⇒ LayoutT bl m α → m (α, Floorplan bl)
runLayoutT (LayoutT m) = do
    (a,fps) ← runWriterT $ flip runReaderT Unspecified m
    return (a, Comb Unspecified fps)
```

```
instance MonadTrans (LayoutT bl)
  where
    lift = LayoutT . lift . lift


class ( IsBlock bl
      , Transformable bl
      , MonadReader Placement m
      , MonadWriter [Floorplan bl] m
      ) ⇒ MonadLayout bl m | m → bl
  where
    block_             :: Width → Height → bl → m ()
    subLayout          :: Placement → m a → m a
    transformFloorplan :: (Floorplan bl → Floorplan bl) → m a → m a


instance ( IsBlock bl
         , Transformable bl
         , MonadReader Placement m
         , MonadWriter [Floorplan bl] m
         ) ⇒ MonadLayout bl m
  where
    block_ x y bl = tell [Block x y bl]

    subLayout pl m = local (const pl) $ censor (λfps → [Comb pl fps]) m

    transformFloorplan trans m = do
        pl ← ask
        censor (λfps → [trans $ Comb pl fps]) m



rightwards, leftwards, upwards, downwards
    :: MonadLayout bl m ⇒ Alignment → m α → m α


rightwards al = subLayout (Row Rightwards al)
leftwards  al = subLayout (Row Leftwards  al)
upwards    al = subLayout (Row Upwards    al)
downwards  al = subLayout (Row Downwards  al)


unplaced :: MonadLayout bl m ⇒ m α → m α
unplaced = subLayout Unspecified


merged :: MonadLayout bl m ⇒ Alignment → Alignment → m α → m α
merged alx aly = subLayout (Merged alx aly)


block :: MonadLayout bl m ⇒ Width → Height → bl → α → m α
block x y bl a = block_ x y bl >> return a
```

```
thinBlock :: MonadLayout bl m ⇒ bl → Width → α → m α
thinBlock bl w a = do
    pl ← ask
    case pl of
         Row Rightwards _ → block w 0 bl a
         Row Leftwards  _ → block w 0 bl a
         Row Upwards    _ → block 0 w bl a
         Row Downwards  _ → block 0 w bl a
         _                → return a

space :: MonadLayout bl m ⇒ Width → α → m α
space = thinBlock spaceBlock

instance MonadLayout bl m ⇒ Transformable (m α)
  where
    flipX   = transformFloorplan flipX
    flipY   = transformFloorplan flipY
    rotate_ = transformFloorplan . rotate_
```