Algebraic Specification and Verification of Processor Microarchitectures

John Robert Matthews
B.S., University of Washington, 1990

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science and Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

October 2000

The dissertation "Algebraic Specification and Verification of Processor Microarchitectures" by John Robert Matthews has been examined and approved by the following Examination Committee:

John Launchbury
Professor
Thesis Research Adviser

Dick Kieburtz
Professor

Dylan McNamee
Assistant Professor

Mary Sheeran
Professor
Chalmers University of Technology

Dedication

To Julie and my parents.

Acknowledgements

I would like to thank my advisor, John Launchbury, for helping me to become a researcher. John has been an excellent teacher and a major source of inspiration in this work. He has given me just the right balance of direction, freedom, and encouragement. Thank you John.

This work was funded by grants from the Air Force Material Command and Intel Corporation, as well as a graduate research fellowship from the National Science Foundation. These endowments helped me to pursue my own research agenda, for which I am grateful. I was also funded by internships at Intel and Microsoft Research (Cambridge), where I gained insight into the pragmatic goals and concerns of industrial verification. I would like to thank Borislav Agapiev and Carl Seger of Intel, as well as Don Syme and Andrew Gordon of Microsoft in giving me these internship opportunities and guidance. I would in addition like to thank the many people at Intel and Microsoft who patiently answered my wide-ranging questions, including Mark Aagaard, Luca Cardelli, Oege de Moor, Robert Jones, Tom Melham, John O'Leary, and Simon Peyton Jones.

I greatly enjoyed my time at the Oregon Graduate Institute, as well as the stimulating discussions and encouragement I received from the members of the PacSoft and Hawk research groups. I spent some exceptional years here.

Byron Cook, Nancy Day, Jeff Lewis, and Thomas Nordin put a lot of time and effort into developing aspects of the Hawk system, which I happily made use of. I fondly remember long talks on aspects of Hawk formalization with Byron Cook, Nancy Day, Sava Krstić and Mark Shields. Their knowledge and insight have improved my thesis.

I would also like to thank the other members of my thesis committee, Dick Kieburtz, Dylan McNamee, and Mary Sheeran, and librarian Julianne Williams for their excellent comments and discussions. I apologize now for any remaining omissions or errors.

I am indebted to my parents Bob Matthews, Elizabeth and Michael O'Connell, and my brother Michael Matthews for the love and encouragement they have shown me for as long as I can remember.

Finally, I would like to thank Julie, the love of my life, for being at my side all of these years, even when I couldn't always be at hers. Thank you for helping me through it all.

Contents

List of Tables

List of Figures

Abstract

Algebraic Specification and Verification

of Processor Microarchitectures

John Robert Matthews

Ph.D., Oregon Graduate Institute of Science and Technology

August, 2000

Thesis Advisor: Dr. John Launchbury

The *Hawk* language is a domain-specific extension of the pure functional language Haskell, and is used to specify and reason about processor microarchitectures at a high level of abstraction. We apply functional language technology and reasoning principles to concisely specify pipelined microarchitectures in Hawk and verify them through a domain-specific *microarchitecture algebra*. We develop a remarkably simple set of local equational laws governing processor components such as register files, bypass logic, and execution units. Many of these laws are verified in *Isabelle*, a higher order logic theorem prover. The laws are used to incrementally simplify a complex pipelined microarchitecture, removing pipeline stages and simplifying control logic, while retaining cycle-accurate behavior with respect to the original pipelined design.

Proving these laws requires defining mutually recursive functions over coinductively defined streams. Such definitions are not directly supported in current theorem provers. We develop a generalization of well-founded recursion, called *Converging Equivalence Relations*, that allows these definitions to be added conservatively in a straightforward and modular fashion.

Chapter 1

Introduction

Modern processor microarchitectures can be incredibly complex. Although exact figures are kept secret, it can safely be said that leading manufacturers employ dozens if not hundreds of design and verification engineers for each new generation of processor. As semiconductor process improvements continue to deliver an exponentially increasing budget of transistors, processor architects are able to employ ever more sophisticated implementation techniques to increase the amount of useful work performed per clock cycle. Some standard examples of performance increasing optimizations are:

- **Pipelining.** Analogous to automobile assembly lines, operations that take more than one clock cycle to complete are often divided into stages. Each stage completes its work in one clock cycle. By connecting the stages with pipeline registers, multiple instances of complex operations can be processed per clock cycle.

- **Superscalar execution.** Multiple instructions are fetched per clock cycle. Duplicated execution units such as ALUs execute the fetched instructions concurrently.

- **Caching.** Long-latency communication between the processor and main memory is minimized by storing past results in local caches for faster access.

- **Out-of-order execution.** Fetched instructions are dynamically analyzed to determine which instructions are independent of each other. Independent instructions are executed according to when a compatible execution unit is available, even though this may cause the operations to be performed in a different order than specified by the program.

- **Speculation.** The results of time-consuming operations are opportunistically predicted. The processor uses the predicted result immediately, and simultaneously starts computing the real result of the operation. The processor then checks whether the prediction is correct once the operation completes. If the prediction is confirmed, the processor has saved time by parallelizing the operation. If the prediction is incorrect, the processor rolls back its internal state and then uses the correct result.

Not only does each of these techniques incur a substantial amount of design complexity, cutting edge processor designs combine them to achieve further speedups. In fact, creating and verifying these designs is a significant proportion of the total microprocessor development lifecycle. As the number of possible gates in future microprocessors increases exponentially, so too does design complexity.

It is now common for a commercial microprocessor design effort to take two years or more, as engineers resolve all of the possible interactions between microarchitectural features while trying to meet performance, area, power, and heat dissipation goals.

Resolving all of these issues while trying to complete the project as quickly as possible almost always results in design defects, some of which may slip through testing efforts and end up in released products. Of course, similar defects routinely occur with large commercial software products. But whereas software faults can be easily fixed by downloading patches through the internet, a microprocessor defect may require the entire device to be replaced. These mistakes can become exceedingly expensive, both financially and in lowered customer confidence. Such mistakes have also become more widely publicized in recent years, as personal computers are increasingly sold to mass consumer markets.

1.1 Hardware description languages

One way to gain intellectual control over design complexity is to employ a formal modeling language. Such a language can provide several benefits. For example, Ashenden[4] notes that assuming the language has appropriate supporting tools, an architect can:

- Describe and understand the required behavior and attributes of a system unambiguously.

- Communicate these requirements to others precisely.
- Test the system by simulating it.
- Formally verify the system with respect to desired properties.
- Automatically synthesize implementations from the description.

Of course, most description languages are not designed to support all of the above activities, at least initially. For example, the VHDL hardware description language[4] has a large set of language features for specifying circuits behaviorally. A user can simulate any behavioral VHDL description, but must describe circuits using a strict subset of these features to automatically synthesize a circuit implementation. On the other hand, low-level languages designed to describe circuits at the gate and transistor level are harder to simulate efficiently.

In practice, a design engineer will typically work with multiple specification languages during a processor development lifecycle. In the early stages, the designer is more concerned with functional correctness and the performance tradeoffs between alternative microarchitectural features at the granularity of individual clock cycles. Thus the design engineer is likely to use a high-level behavioral specification language, such as behavioral VHDL, or even C. As the overall design is solidified, lower level structural considerations, such as size and layout constraints, power consumption budgets, and sub-clock-cycle timing issues often encourage or require the engineer to develop circuit designs that can be directly synthesized and analyzed at the gate or transistor level.

1.1.1 Goals of the Hawk language

At the Oregon Graduate Institute we have been interested in developing high-level domain specific programming languages based on structuring principles derived from typed functional programming languages. In particular, the *Hawk* project has been developing a behavioral specification language for processor microarchitectures. Our goal is to build a language that lets architects specify designs at a higher level of abstraction than can be done with current behavioral hardware specification languages. To achieve this we intend to use language features that promote *concision*, *modularity*, and *reusability* in specifications.

- **Concision.** Just as a program written in a higher level language such as C is easier for humans to understand and modify than the same program written in assembly language, so too do microarchitectures become more comprehensible as specifications are made more concise and abstract. Ideally we would like our specification language to be as concise as the high-level block diagrams that architects currently use to express microarchitectures.

- **Modularity.** Given the number of people required to design modern processor microarchitectures, it is essential to be able to decompose a large specification into separate units, with well-defined interfaces between them. In this way individual architects can concentrate on a portion of the overall microarchitecture, without having to understand the entire design in full detail.

- **Reusability.** Once a specification language has the ability to separate design elements into modular units, a natural next step is to try to reuse commonly occuring design units by defining them once and then referring to the definition at each point of use. By eliminating redundant definitions, the overall size of the specification is reduced, and defects caused by creating incompatible versions of the same design element are prevented.

However, we don't want our specification language to be so abstract that it is not executable. To gain confidence in a design's correctness and evaluate performance tradeoffs an architect may need to simulate a microarchitecture on a wide variety of programs. It is not uncommon for a microprocessor simulator to execute billions of instructions on a given design.

In addition to concrete simulation, we would also like to simulate microarchitectures in Hawk symbolically. A symbolic simulator allows the user to execute a design with some of the inputs given as symbolic variables (or more generally expressions), rather than as concrete values. The simulator then executes the design with the symbolic inputs and returns the result as a symbolic expression. In this way a single symbolic test run can replace a whole family of concrete test runs. A good introduction to symbolic simulation techniques for processors is given by Moore[67], who uses the ACL2 theorem prover to symbolically simulate a small processor at the instruction set architecture level. Symbolic simulation can sometimes detect errors simply because the returned expression "looks strange", i.e. is much larger or more complex than what was expected. This strategy was used by Greve[31] to detect microcode errors in a direct execution Java processor. Day, Lewis, and Cook[19] have developed a version of Hawk that supports symbolic simulation and have used it to symbolically simulate the data flow of a superscalar out-of-order microarchitecture.

To gain even more confidence in the correctness of a Hawk specification an architect should be able to turn to *formal verification*, where a mathematical proof demonstrates that a design satisfies desired correctness properties on all possible inputs. Since the design being verified can be quite large, this approach only becomes practical when the proof is carried out with the help of automated tools, such as model checkers and theorem provers. Constructing proofs requires formalizing both the design and the underlying specification language in some mathematical logic, such as set theory or higher order logic. This is not a trivial endeavor, and specification languages with complex or ill-defined semantics can substantially increase the amount of human and machine time necessary to complete the proof.

1.2 Thesis statement

Hawk was created as a typed functional programming language in order to provide a good balance between abstraction and expressiveness, executability, and ease of formal reasoning. In particular, this dissertation aims to show that:

- The concepts underlying lazy functional programming languages, particularly Haskell and its Hawk extensions, allow one to specify microarchitectures concisely, modularly, and reusably, while retaining the ability to simulate them on concrete test cases.

- Using equational reasoning principles, one can develop a *microarchitecture algebra*, whose laws enjoy the same degree of concision, modularity, and reusability as the microarchitecture specification.
- Such algebraic laws can be used to verify the correctness of pipelined microarchitectures.
- The Hawk specification language can be naturally formalized in higher order logic, and thus verification steps can be checked by a theorem prover.

This thesis can be thought of as a case study supporting a larger agenda: To demonstrate that the equational reasoning principles underlying lazy functional languages, and specifically the Haskell programming language, provide a good foundation for developing domain-specific algebras. The hope is that such algebras increase one's understanding of the domains, and can be used to formally verify desired properties of specifications.

1.3 Synopsis

Part of the content of this thesis is made up of re-edited and expanded versions of three published papers and a technical report, all written primarily by this author. These papers introduce Hawk as a specification language[55], describe how algebraic reasoning can be used to simplify and verify pipelined microarchitectures[53, 54], and show how to define recursive functions, such as Hawk circuits, over coinductive types[52].

Accordingly, we begin the dissertation by introducing Hawk as a microarchitecture specification language embedded within Haskell. We then state equational laws that hold of microarchitectural components, such as register files and ALUs, and use them to incrementally simplify a pipelined microarchitecture. Finally, we formalize a subset of Hawk in higher order logic and prove a representative set of these microarchitecture laws, using a combination of equational reasoning and induction over time.

The definition of mutually recursive functions over infinite streams is the most challenging aspect of Hawk's formalization, since such definitions are not directly supported in current theorem provers. We develop a generalization of well-founded recursion, called

Converging Equivalence Relations, that allows these definitions to be added conservatively in a straightforward and modular fashion.

The remaining chapters of this thesis are as follows:

Chapter II: Introduction to Hawk

This chapter introduces Hawk as a specification language. We introduce a simple pipelined microarchitecture and specify it first in Hawk at the register transfer level (RTL) and then with *transactions*, an abstract datatype for representing the complete microarchitectural state associated with an instruction. We show that the language features of Hawk combined with transactions as a structuring principle lead to a concise and understandable specification.

Chapter III: Microarchitecture algebra

Next, we informally introduce our algebra of microarchitectural components by describing the components that comprise a more complex reference architecture than the one introduced in Chapter 2. We describe how these components are modeled in Hawk and state the laws that hold among them.

Several of the laws contain *projection* circuits. Projections are not used in either the pipelined or the reference microarchitectures, but are instead artifacts of the verification process. We motivate the usefulness of projections, and describe the conditions under which they can appear in microarchitecture laws.

Once the necessary laws have been introduced, we show how they can be used to simplify the pipelined microarchitecture. This simplification is presented graphically.

Chapter IV: Formalizing Hawk in higher order logic

In this chapter we introduce Higher Order Logic (HOL) and the *Isabelle* theorem prover briefly and informally. We use HOL to formalize Hawk and the microarchitecture algebra, and Isabelle to check the proofs. Since Hawk is a purely applicative functional language, many aspects of the language can be modeled directly in higher order logic itself. However, dealing with recursive Hawk definitions is more difficult. The standard semantics for

Hawk is domain theoretic, with recursive definitions modeled by least fixpoints. Although Isabelle has an object logic (HOLCF) that provides some support for reasoning about domains, there is much more support for "pure" HOL. For example, there is no syntactic support in HOLCF for pattern-matching function definitions or pointed numeric domains. We thus focus on techniques for modeling Hawk directly in HOL.

There is no natural "information order" among elements in pure HOL, and so there is no notion of a least fixpoint. However, it turns out that well-formed recursive Hawk definitions have *unique* fixpoints, and can therefore be uniquely defined using Hilbert's choice operator. It is a well known result of topology that unique fixpoints can be found for *contracting* functions in *complete metric spaces*. Intuitively, a metric space is a set of elements and an associated *distance metric* over pairs of elements. The distance metric returns a real-valued number indicating how far apart the two elements are. A contracting function over this metric space, when applied to each of a pair of elements, returns a corresponding pair of elements that is "closer" to each other than the original elements are. Banach's theorem states that contracting functions do in fact have unique fixpoints.

It is possible to define suitable distance metrics for Hawk streams, and show that recursive Hawk definitions over these streams are contracting functions. However, this often requires reasoning about division and exponentiation over real-valued domains, which relatively few theorem provers support well. Instead we adopt a different approach.

Chapter V: Converging equivalence relations

We develop an alternative framework, called *Converging Equivalence Relations* (CERs), for proving the uniqueness of fixpoint definitions. We develop analogs of metric spaces and contracting functions that do not require the use of continuous mathematics. Instead, reasoning proceeds by well-founded induction over discrete domains such as the natural numbers, which are well supported by all of the HOL-based theorem provers.

This chapter describes CERs with proofs of the key results. We demonstrate that this technique can be mostly automated by Isabelle's higher-order tableau proof package.

Chapter VI: Verifying the microarchitecture laws

In this chapter we develop some techniques to simplify the proofs of the individual laws of Chapter 3, and use them to verify representative examples. We first develop a simple theory of transactions, and make the somewhat surprising observation that although the type system of Hawk is very useful in catching errors when constructing Hawk specifications, it can be annoyingly restrictive when verifying laws about transaction fields. The statements of these theorems quantify over all of the fields in a transaction, which violates the HOL restriction that all quantifiers must range over elements of the same type. We develop a mechanism of *first class field names* to overcome this difficulty.

We use a combination of inductive reasoning over time and first class field names to prove two representative microarchitecture laws: a commutativity law between ALU and delay components, and a law that allows one to remove bypass circuits connected to register files.

Chapter VII: Retrospective

We conclude by analyzing the strengths and weaknesses of Hawk we encountered during the course of the dissertation. In particular, we discuss the relative merits of the functional basis of Hawk, the use of transactions, and the value of algebraic reasoning in the context of the Isabelle theorem prover.

We also discuss the usefulness of defining functions by converging equivalence relations, compared to defining them co-recursively. The CER framework provides a general method of defining recursive functions over a wide range of types, including coinductively defined types such as infinite lists and trees. The dissertation concludes by outlining future research directions.

Chapter 2

Introduction to Hawk

The *Hawk* language is designed for building executable specifications of processor microarchitectures. Currently Hawk is an embedded language (i.e. a set of libraries) within Haskell, a strongly-typed functional language with powerful abstraction capabilities, such as lazy (demand-driven) evaluation, first-class functions, and parametric polymorphism [35] [76].

2.1 The Hawk library

We start with a simple example that introduces several functions used in later examples. Consider the resettable counter circuit of Figure 2.1.

The *reset* wire is Boolean valued, while the other wires are integer valued. Of course, in silicon, integer-valued wires are represented by a vector of Boolean wires, but as a design abstraction, a Hawk user may choose to use a single wire. The circuit counts (and outputs) the number of clock cycles since *reset* was last asserted.

2.1.1 Signals

Notice that there is no explicit clock in the diagram. Rather, each wire in the diagram carries a *signal* (integer or boolean valued) which is an implicitly clocked value. The output of a circuit only changes between clock cycles. We build signals using an abstract type constructor called `Signal`. As a mental model we could think of a value of type `Signal a` as a function from non-negative integers to values of type `a`, as is often done in the hardware verification community[62, 92].

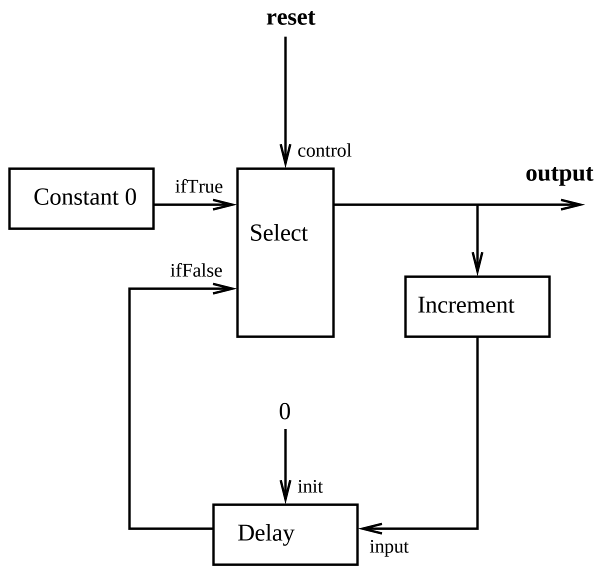


Figure 2.1: Resettable Counter. A simple circuit that counts the number of clock cycles between reset signals.

```
type Signal a = (Int -> a)
```

We can sample a signal s at a given clock cycle n simply by evaluating s applied[1] to n. Alternatively a signal could be thought of as an infinite stream of values $\langle x_0,\ x_1,\ x_2,\ \ldots\rangle$. Clearly the two views are interchangeable. In either case circuits are represented as functions from signals to signals.

In the resettable counter example above, the *constant 0* circuit outputs zero on every clock cycle. The *select* component chooses between its inputs on each clock cycle depending on the value of *reset*. If *reset* is asserted on a given cycle (has value *true*), then the output is equal to *select*'s top input, in this case zero. If *reset* is not asserted, then its output is the value of its bottom input. In either case, *select*'s output is the output of the entire circuit, as well as the input to the *increment* component, which simply adds 1 to its input. The output of *increment* is fed into the *delay* component. A delay component outputs whatever was on its input in the previous clock cycle: it "delays" its input by one cycle (delay circuits occur often enough that we draw them specially, as shaded horizontal or

[1]Function application in Hawk is written using juxtaposition, so that f applied to x and y is written as $f\ x\ y$

vertical bars). However, on the first clock cycle of the simulation there is no previous input, so on the first cycle *delay* outputs whatever is on its *init* input, which is zero in this circuit.

2.1.2 Components

The components used in the resettable counter are trivial examples of the sorts of things provided by the Hawk library, but let's look at a specification of each component in turn.

The simplest component is `constant`

```
constant :: a -> Signal a
```

The constant function takes an input of any type `a`, and returns an output of type `Signal a`, that is, a sequence of values of type `a`. For every clock cycle, `(constant x)` always has the same value `x`. Functions such as `constant` that can operate over more than one type are called *polymorphic*.

The next component is `select`:

```
select :: Signal Bool -> Signal a -> Signal a -> Signal a
```

This declares `select` to be a function. In a Hawk declaration, anything to the left of an arrow is an argument to a function. Thus, the expression `(select bs xs ys)`, where `bs` is a Boolean signal, and `xs` and `ys` are signals of type `a`, will return an output signal of type `a`. The values of the output signal are drawn from `xs` and `ys`, decided each clock tick by the corresponding value of `bs`. For example, if

```
bs = <True,False,True,False,...>,
xs = <x1,x2,x3,x4,...>,
ys = <y1,y2,y3,y4,...>
```

then `(select bs xs ys)` is equal to the signal `<x1,y2,x3,y4,...>`.

Hawk treats functions as first-class values, allowing them to be passed as arguments to other functions or returned as results. First-class functions allow us to specify a generic `lift` primitive, which "lifts" a normal function from type `a` to type `b` into a function over the corresponding signal types:

```
lift :: (a -> b) -> Signal a -> Signal b
```

The expression (lift f xs), where xs = <x1,x2,x3,...>, is equal to the signal <f x1, f x2, f x3, ...>.

The increment component is defined in terms of lift:

```
increment :: Signal Int -> Signal Int
increment xs = lift (+ 1) xs
```

Given the xs input signal, increment adds one to each component of xs and returns the result.

The delay component is more interesting:

```
delay :: a -> Signal a -> Signal a
```

This function takes an initial value of type a, and an input signal of type Signal a, and returns a value of type Signal a (the input arguments are in reverse order from the diagram). At clock cycle zero, the expression (delay initVal xs) returns initVal. Otherwise the expression returns whatever value xs had at the previous clock cycle. This function can thus propagate values from one clock cycle to the next.

2.1.3 Using the components

Once we have defined primitive signal components like the ones above, we can define the resettable counter:

```
resetCounter :: Signal Bool -> Signal Int
resetCounter reset = output
  where next   = delay 0 (increment output)
        output = select reset (constant 0) next
```

The resetCounter definition takes reset as a Boolean signal, and returns an integer signal. The reset signal is passed into select. On every clock cycle where reset returns True, select outputs 0, otherwise it outputs the result of the next signal. On the first clock cycle next outputs 0, and thereafter outputs the result of whatever (increment output) was on the previous clock cycle. The output of the whole circuit is the output of the select function, here called output. Notice that output is used twice in this function: once as the input to `increment`, and once as the result of the entire function. This corresponds to the fact that the `output` wire in Figure 1 is split and used in two places. Whenever a wire is duplicated in this fashion, we must use a `where` statement in Hawk to name the wire.

2.1.4 Recursive definitions

There is something else curious about the `output` variable. It is being used recursively in the same place it is being defined! Most languages only allow such recursion for functions with explicit arguments. In Hawk, one can also recursion to define data-structures and functions with implicit arguments, such as the one above.

If we didn't have this ability, we would have had to define `resetCounter` as follows:

```
resetCounter reset = output
  where next t   = (delay 0 (increment output)) t
        output t = (select reset (constant 0) next) t
```

Every time we have a cycle in a circuit, we would have to create a local recursive function, passing an explicit time parameter. This breaks the abstraction of the `Signal` ADT. In fact, in the real implementation of signals, we don't use functions at all. We use infinite lists instead. Each element of the list corresponds to a value at a particular clock cycle; the first list element corresponds to the first clock cycle, the second element to the second clock cycle, and so on. By storing signals as lazy lists, we compute a signal value at a given clock cycle only once, no matter how many times it is subsequently accessed.

Haskell allows recursive definitions of abstract data structures because it is a lazy language, that is, it only computes a part of a data structure when some client code demands its value. It is lazy evaluation that allows Haskell to simulate infinite data structures, such as infinite lists.

2.1.5 Other embedded Haskell languages

Hardware domains

The Hawk team is not the first to take advantage of Haskell as a platform for embedding domain specific languages, or even languages for modeling hardware. For example, O'Donnell[70] has developed a Haskell library called Hydra that models hardware gates at several levels of abstraction, ranging from implementations of gates using CMOS and NMOS pass-transistors, up to abstract gate representations using lazy lists to denote time-varying values. Hydra has been used to teach advanced undergraduate courses on computer design, in which students use Hydra eventually to design and test a simple microprocessor. Hydra is similar to Hawk in many ways, including the use of higher-order functions and lazy lists to model signals. However, Hydra does not allow users to define more structured signal types, such as signals of integers or signals of transactions. In Hydra, these composite types have to be built up as tuples or lists of Boolean signals. While this limitation does not cause problems in an introductory computer architecture course, structured signal types significantly reduce specification complexity for more realistic microprocessor specifications.

More recently, the Lava hardware description language has been designed. It also models gate and word level hardware circuits within Haskell. The original version of Lava[9] modeled circuits with a special *monadic* syntax, however a later version[14] defines circuits using standard Haskell expression form, in the same manner as Hawk. Modern Lava has many other similarities to Hawk: Both model signals as first class entities, use polymorphism and higher-order functions to model generic wiring patterns, and model circuits with feedback as recursively-defined signals. Lava is discussed in more detail in Section 2.4.

MHDL[6] is a hardware description language for describing analog microwave circuits, and includes an interface to VHDL. Though it tackles a very different part of the hardware design spectrum, like Hawk, MHDL is essentially an extended version of Haskell, although it is not technically an embedded language. The MHDL extensions have to do with physical units on numbers, and universal variables to track frequency and time etc.

Other domain specific languages

Haskell has successfully been used to specify other domains. For example, Haskell compared favorably in an experiment comparing several prototyping languages[34]. The application domain involved modeling the *Geometric Region Server* module, which tracks the regions surrounding ships and planes in a military theatre. The module is required to answer such questions as when an enemy plane will enter a friendly ship's weapons range, or whether a plane has entered a commercial airspace corridor. Experts in each of several languages including Haskell, C++, Awk, and Griffin wrote a prototype program based on the same requirements document. The Haskell solution was considered the most concise and understandable of all the submitted entries. The authors claim their major success factors were: their heavy use of higher-order functions, Haskell's simple syntax, and the availability of powerful list-manipulating primitives in the standard Haskell library.

Fran[23] is a Haskell library that models interactive multimedia animations. The authors provide ADTs for time-varying behaviors, events, and interactions between behaviors and events. Unlike Hawk, Fran's model of time is continuous. Also, a Fran function can examine the values of future events, while Hawk signals only depend on current and past signal values. This non-monotonicity of time in Fran requires a more sophisticated *time-interval* analysis than is required for Hawk.

2.2 A simple microprocessor

In the microarchitecture domain, the Hawk libraries make essential use of Haskell's features. As a test of Hawk's capabilities, the Hawk team has specified and simulated several versions of the DLX microprocessor described in Hennessy and Patterson's widely used textbook[33]. The Hawk team chose to model the DLX because it is well known, and has excellent tool support. Several DLX simulators exist, as well as a version of the Gnu C compiler that generates DLX assembly instructions. The processor includes the most common instructions found in commercial RISC processors.

The DLX architecture is too complex to explain in fine detail in an introductory chapter. Instead, for pedagogical purposes we show how to specify a simple microprocessor called SHAM (Simple HAwk Microprocessor). We begin with the simplest possible SHAM architecture (unpipelined), and then add features: pipelining, and a memory-cache. A corresponding annotated Hawk specification of the DLX itself can be found at the Hawk web page[44].

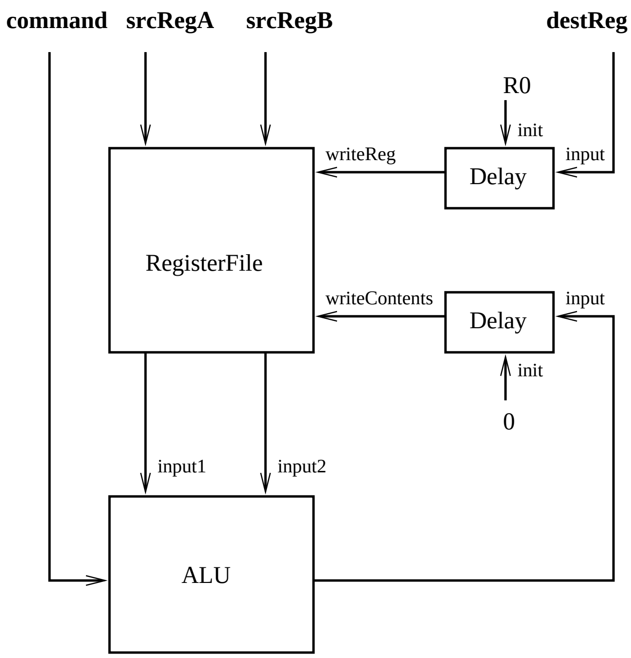


Figure 2.2: Unpipelined version of SHAM.

The unpipelined SHAM diagram is shown in Figure 2.2. The microprocessor consists of an ALU and a register file. The ALU recognizes three operations: `ADD`, `SUB`, and `INC`. The `ADD` and `SUB` operations add and subtract, respectively, the contents of the two ALU inputs. The `INC` operation causes the ALU to increment its first input by one and output the result. The register file contains eight integer registers, numbered `R0` through `R7`. Register `R0` is hardwired to the value zero, so writes to `R0` have no effect. The register file has one write-port and two read-ports. The write-port is a pair of wires; the register to update, called *writeReg*, and the value being written, called *writeContents*. The input to each read-port is a wire carrying a register name. The contents of the named read-port registers are output every cycle along the wires *contentsA* and *contentsB*. If a register is written to and read from during the same clock cycle, the newly written value is reflected in the read-port's output, at least abstractly. This is consistent with the behavior of most modern microprocessor register files.

SHAM instructions are provided externally; in our drive for simplicity there is no notion of a program counter. Each instruction consists of an ALU operation, the destination register name, and the two source register names. For each instruction the contents of the two source registers are loaded into the ALU's inputs, and the ALU's result is written back into the destination register.

2.2.1 Unpipelined SHAM specification

Let us assume we have already specified the register file and ALU, with the signatures below:

```
data Reg = R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7

regFile :: Signal Reg -> Signal Reg ->
           (Signal Reg, Signal Int) ->
           (Signal Int, Signal Int)

data Cmd = ADD | SUB | INC

alu :: Signal Cmd -> Signal Int -> Signal Int -> Signal Int
```

The `regFile` specification takes two read-port inputs, a write-port input, and returns the corresponding read-port outputs. The `alu` specification takes a command signal and two input signals, and returns a result signal. Given these signatures and the previous definition of `delay`, it is easy in Hawk to specify an unpipelined version of SHAM:

```
sham1 :: (Signal Cmd,Signal Reg,Signal Reg,Signal Reg) -> (Signal Reg,Signal Int)

sham1 (cmd,destReg,srcRegA,srcRegB) = (destReg',aluOutput')
  where
  (aluInputA,aluInputB) = regFile srcRegA srcRegB (destReg',aluOutput')
  aluOutput             = alu cmd aluInputA aluInputB
  aluOutput'            = delay 0 aluOutput
  destReg'              = delay R0 destReg
```

The definition of `sham1` takes a tuple of signals representing the stream of instructions, and returns a pair of signals representing the sequence of register assignments generated by the instructions. The first three lines in the body of `sham1` read the source register values from the register file and perform the ALU operation. The next two lines delay the destination register name and ALU output, in effect returning the values of the previous clock cycle. The delayed signals become the write-port for the register file. It is necessary to delay the write-port since modifications to the register file logically take effect for the next instruction, not the current one.

2.2.2 Pipelining

Suppose we wanted to increase SHAM's performance by doubling the clock frequency. We will assume that, while `sham1` could perform both the register file and ALU operations within one clock cycle, with the increased frequency it will take two clock cycles to perform both functions serially. We use pipelining to increase the overall performance. While the ALU is working on instruction n, the register file will be writing the result of instruction $n-1$ back into the appropriate register, and simultaneously reading the source registers of instruction $n+1$.

But now consider a sequence of instructions such as:

```
R2 <- R1 ADD R3
R4 <- R2 SUB R5
```

When the `ADD` instruction is in the ALU stage, the `SUB` instruction is in the register-fetch stage. But one of the registers that is being fetched (`R2`), has not been written back into the register file yet, because the ALU is still calculating the result. The `SUB` instruction will read an out-of-date value for `R2`. This is an example of a *data hazard*, where naive pipelining can produce a result different from the unpipelined version of a microprocessor. To resolve this hazard, we will first add *bypass logic* to the pipeline. Later we will abstract away from this added complexity.

Figure 2.3 contains the diagram of a pipelined version of SHAM with bypass logic. By the time the source operands to the `SUB` instruction (`R2` and `R5`) are ready to be input into the ALU, the up-to-date value for `R2` is stored in the delay circuit between the ALU and the register file's write-port. The bypass logic uses this stored value of `R2` as the input to the ALU, rather than the out-of-date value read from the register file. The bypass logic examines the incoming instructions to determine when this is necessary. The following code contains the Hawk specification:

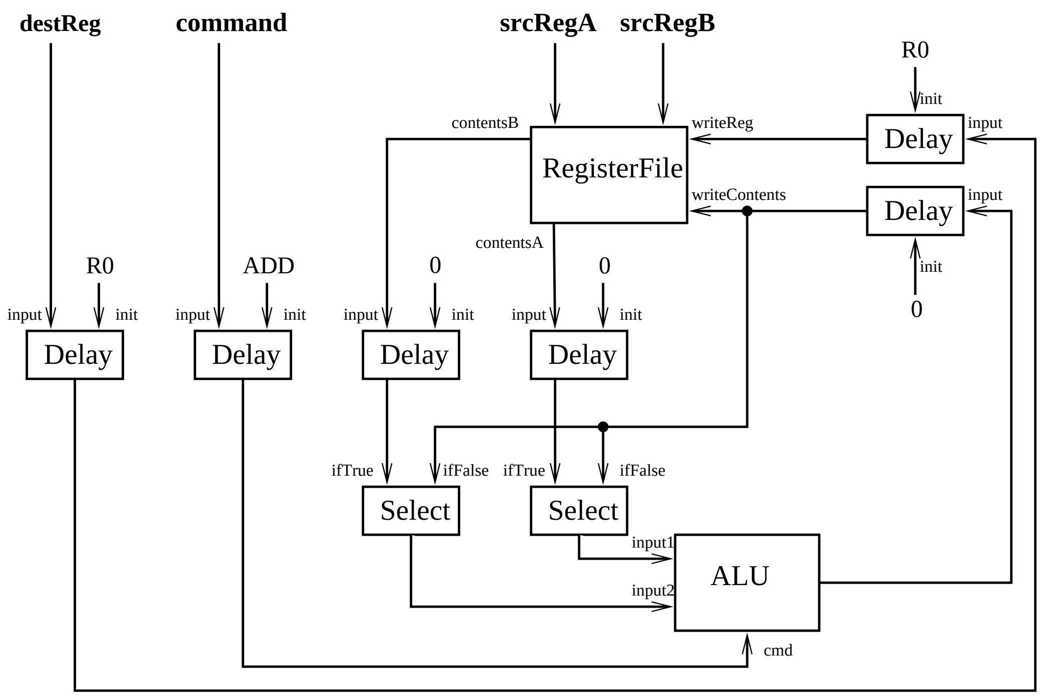


Figure 2.3: Pipelined SHAM. Since the register file and the ALU each now take one clock cycle to complete, we now need extra *Delay* circuits. The *Delay* circuits in turn require us to add *Select* circuits to act as bypasses. The logic controlling the *Select* circuits is not shown.

```
sham2 :: (Signal Cmd,Signal Reg,Signal Reg,Signal Reg) -> (Signal Reg,Signal Int)

sham2 (cmd,destReg,srcRegA,srcRegB) = (destReg'',aluOut')
  where
  (valueA,valueB) = regFile srcRegA srcRegB (destReg'',aluOut')

  valueA'    = delay 0 valueA
  valueB'    = delay 0 valueB
  destReg'   = delay R0 destReg
  cmd'       = delay ADD cmd

  aluInputA  = select validA valueA' aluOut'
  aluInputB  = select validB valueB' aluOut'

  aluOut     = alu cmd' aluInputA aluInputB

  aluOut'    = delay 0 aluOut
  destReg''  = delay R0 destReg'

  --- Control logic ---

  validA     = delay True (noHazard srcRegA)
  validB     = delay True (noHazard srcRegB)

  noHazard :: Signal Reg -> Signal Bool
  noHazard srcReg = sigOr (sigEqual destReg' (constant R0))
                          (sigNotEqual destReg' srcReg)
```

The data flow portion of the code is grouped according to pipeline stages:

- The first line after the `where` keyword reads the contents of the source registers from the register file.
- The next four lines delay the source register contents, the ALU command, and the destination register name by one cycle.
- The two `select` commands decide whether the delayed values should be bypassed. The decision is made by the Boolean signals `validA` and `validB`, which are defined in the control logic section.

- The next line performs the ALU operation.
- The last two lines in the data-flow section delay the ALU result and the destination register. The delayed result, called `aluOut'`, is written back into the register file in the register named by `destReg''`, as indicated in the first two lines of the section.

The control logic section determines when to bypass the ALU inputs. The signals `validA` and `validB` are set to `True` whenever the corresponding ALU input is up-to-date. The definition of these signals uses the function `noHazard`, which tests whether the previous instruction's destination register name matches a source register name of the current instruction. If they do, then the function returns `False`. The exception to this is when the destination register is `R0`. In this case the ALU input is always up-to-date, so `noHazard` returns `True`.

2.2.3 Transactions

The definition of `sham2` highlights a difficulty of many such specifications. Although the data flow section is relatively easy to understand, the control logic section is far from satisfactory. In fact, it often takes nearly as many lines of Hawk code to specify the control logic as it does to specify the data flow, and mistakes in the control logic may not be easy to spot. We need a more intuitive way of defining control logic sections in microprocessors.

We use a notion of *transactions* within Hawk to specify the state of an entire instruction as it travels through the microprocessor (similar in spirit to Aagaard and Leeser [1]). A transaction holds an instruction's source operand values, the ALU command, and the destination operand value. Transactions also record the register names associated with the source and destination operands:

```
data Transaction = Trans DestOp Cmd SrcOp SrcOp

type DestOp  = Operand
type SrcOp   = Operand
type Operand = (Reg,Value)
```

```
data Value = Unknown | Val Int
```

An operand is a pair containing a register and its value. Values can either be "unknown" or they can be known, e.g. `Val 7`.

For example, the instruction `(R3 <- R2 ADD R1)`, when it has completed, would be encoded as shown below (assume that register `R2` holds the value `3`, and `R1` holds `4`):

```
Trans (R3,Val 7) ADD (R2,Val 3) (R1,Val 4)
```

This expression states that register `R3` should be assigned the value `7` as a result of adding the contents of register `R2` and `R1`.

Not all of the register values in a transaction are known in the early stages of the pipeline. When a register name does not have an associated value yet, it is assigned the value `Unknown`. For example, if the above instruction had not reached the ALU stage yet, then the corresponding transaction would be:

```
Trans (R3,Unknown) ADD (R2,(Val 3)) (R1,(Val 4))
```

Figure 2.4 shows how a transaction's values are filled in as it flows through the pipeline.

2.2.4 Transaction structure

In general, the `Transaction` datatype contains four subfields. The first field holds the destination register name and its current state. The *state* of a register indicates the current value for the register at a given stage of the pipeline. Possible state values are `Unknown`, or `(Val k)`. The second field is the instruction's ALU operation, in this case the `ADD` command. The third and fourth fields hold the source operand register names and their corresponding states. In this example, it holds the names and states for the source operands `R2` and `R1`. If an instruction has less than two source operands, the extra operand fields are set to a default value of `(R0, Val 0)`.

The instruction `(R3 <- R2 ADD R1)`, before it enters the SHAM pipeline, is encoded as the transaction:

```
Trans (R3,Unknown) ADD (R2,Unknown) (R1,Unknown)
```

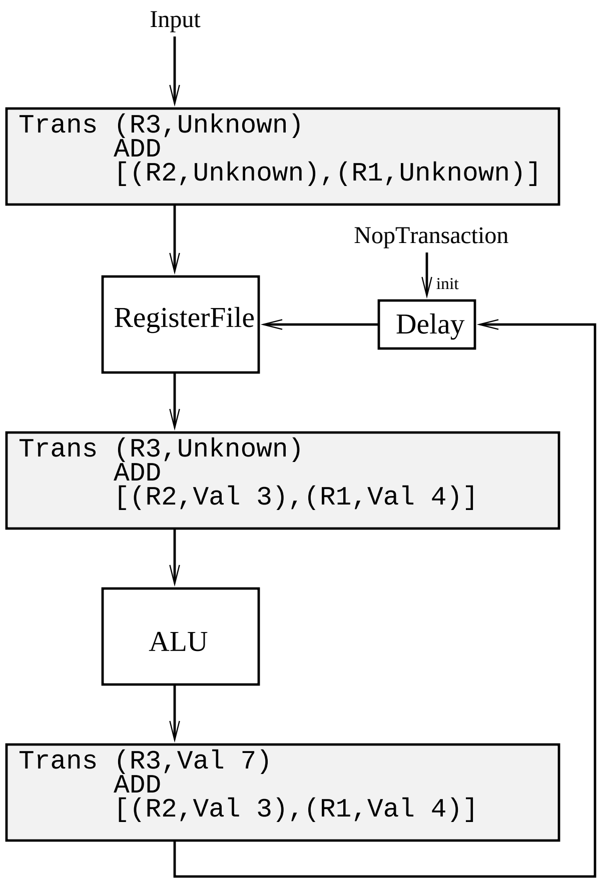


Figure 2.4: A transaction as it flows through the pipeline. As the transaction progresses, its operands become more refined.

At this point, none of the register values are known.

2.2.5 Changes to handle transactions

We change the `regFile` and `alu` functions so that they take and return transactions:

```
regFile :: Signal Transaction ->
           Signal Transaction ->
           Signal Transaction

alu :: Signal Transaction ->
       Signal Transaction
```

Because the register file needs to both write new values to the CPU registers and read values from them, the `regFile` function takes a *read transaction* and a *writeback* *transaction* as inputs. The function first examines the destination register field of the writeback transaction and updates the corresponding register in the register file. It then outputs the read transaction, modified so that all of the source register fields contain current values from the register file. For example, suppose `regFile` is applied to the completed write-transaction (the second source operand is not used here):

```
Trans (R1,Val 4) INC (R1,Val 3) (R0,Val 0)
```

and uses as its read transaction

```
Trans (R3,Unknown) ADD (R2,Unknown) (R1,Unknown)
```

If we further assume that register `R1` is assigned `20` and `R2` is assigned `3` before `regFile`'s application, then `regFile` will update `R1` to contain `4` from the writeback transaction, and will output a new transaction that is identical to the read transaction, except that all of the source registers have been assigned current values from the register file:

```
Trans (R3,Unknown) ADD (R2,Val 3) (R1,Val 4)
```

The revised `alu` function takes a transaction whose source operands have values, performs the appropriate operation, and outputs a modified transaction whose destination field has been filled in. Thus if the `ADD` transaction above were given to `alu`, it would return:

```
Trans (R3,Val 7) ADD (R2,Val 3) (R1,Val 4)
```

2.2.6 Unpipelined SHAM

Using transactions, the unpipelined version of SHAM is even easier to specify than it was before.

```
sham1Trans :: Signal Transaction ->
              Signal Transaction
sham1Trans instr =  aluOutput'
  where
    aluInput = regFile instr aluOutput'
    aluOutput = alu aluInput
```

```
    aluOutput' = delay nop aluOutput

nop = Trans (R0,Val 0) ADD (R0,Val 0) (R0,Val 0)
```

But the real benefit of transactions comes from specifying more complex micro-architectures, as we shall see next.

2.2.7 SHAM2 with transactions

Transactions are designed to contain the necessary information for concisely specifying control logic. The control logic needs to determine when an instruction's source operand is dependent on another instruction's destination operand. To calculate the dependency, the source and destination register names must be available. The transaction carries these names for each instruction. Because of this additional information, bypass logic is easily modeled with following combinator:

```
bypass :: Signal Transaction ->
          Signal Transaction ->
          Signal Transaction
```

At any cycle, the `bypass` function usually just outputs its first argument. Sometimes, however, the second argument's destination operand name matches one or more of the first argument's source operand names. In this case, the matching source operand's state values are updated to equal the destination operand state value. The updated version of the first argument is then returned.

So if at clock cycle n the first argument to bypass is:

```
Trans (R4,Unknown) ADD (R3,Val 12) (R2,Val 4)
```

and the second argument at cycle n is:

```
Trans (R3,Val 20) SUB (R8,Val 2) (R11,Val 10)
```

then because `R3` in the second transaction's destination field matches `R3` in the first transaction's source field, the output of `bypass` will be an updated version of the first transaction:

```
Trans (R4,Unknown) ADD (R3,Val 20) (R2,Val 4)
```

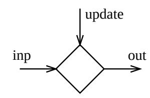


Figure 2.5: bypass circuit

One special case to `bypass`'s functionality is when a source register is `R0`. Since `R0` is a constant register, it does not get updated.

Bypasses arise frequently enough in pipeline block-level diagrams that we draw them specially, as diamonds with the `update` input (i.e. the second argument) connected to either the top or the bottom, as shown in Figure 2.5.

The pipelined version of SHAM with bypass logic is now straightforward. Notice that no explicit control logic is needed, as all the decisions are taken locally in the bypass operations.

```
SHAM2Trans :: Signal Transaction -> Signal Transaction

SHAM2Trans instr = aluOutput'
  where
  readyInstr  = regFile instr aluOutput'
  readyInstr' = delay nop readyInstr
  aluInput    = bypass readyInstr' aluOutput'
  aluOutput   = alu aluInput
  aluOutput'  = delay nop aluOutput
```

The first line takes `instr` and fills in its source operand fields from the register file. The filled-in transaction is delayed by one cycle in the second line. In the third line `bypass` is invoked to ensure that all of the source operands are up-to-date. Finally the transaction result is computed by `alu` and delayed one cycle so that the destination operand can be written back to the register file.

2.2.8 Hazards

There are some microprocessor hazards that cannot be handled through bypassing. For example, suppose we extended the SHAM architecture to process load and store instructions:

```
R3 <- MEM[R2]
MEM[R5] <- R2
```

The first instruction above is a load instruction; it loads the contents of the address pointed to by `R2` into `R3`. The second instruction is a store; it stores the contents of `R2` into the address pointed to by `R5`. A block diagram of the extended SHAM architecture is shown in Figure 2.6. There is now a load/store pipeline stage after the ALU stage. However, this introduces a new problem. Suppose SHAM executes the following two instructions in sequence:

```
R2 <- MEM[R1]
R4 <- R2 ADD R3
```

These two instructions have a data hazard, just as before, but we can not use bypassing to resolve it. Bypassing depends on having a value to bypass at the *beginning* of a clock cycle, but `R2`'s value won't be known until the end of the cycle, after the memory contents have been retrieved from the memory cache. To resolve this hazard, we have to *stall* the pipeline at the register-fetch stage. When the first instruction has reached the end of the ALU stage, the second instruction will have reached the end of the register-fetch stage. At this point the delay circuits between the register-fetch stage and the ALU stage are overridden; on the next clock cycle they instead output the equivalent of a no-op instruction. The register-fetch stage itself re-reads the second instruction on the next clock cycle. In effect, the pipeline stall inserts a no-op instruction between the two instructions involved in the hazard:

```
R2 <- MEM[R1]
NOP
R4 <- R2 ADD R3
```

Now when the `ADD` instruction is about to be processed by the ALU, the load instruction has already completed the memory stage. `R2`'s value is held in the pipeline registers after the memory stage, so bypass logic can be used to bring the ALU's input up-to-date. In order to stall correctly, we have to re-read the second instruction. Thus stalling reduces the performance of the pipeline.

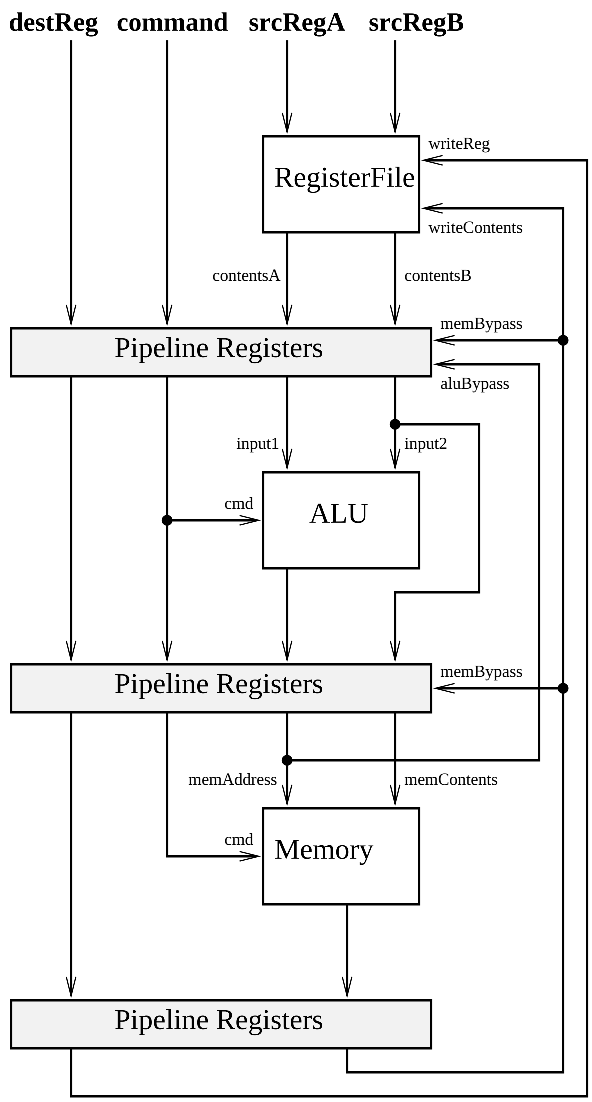


Figure 2.6: Block diagram of extended SHAM pipeline. Each *Pipeline Register* circuit is made up of multiple *Delay* and *Select* circuits. The *Select* circuits are used for bypassing, ensuring that the source operands are up-to-date.

2.2.9 Hawk specification of extended SHAM

In this section we will give more evidence of the simplifying power of transactions by specifying the extended SHAM architecture. The load/store extension significantly complicates the control logic for the SHAM architecture. We shall see that transactions hold up well when we must add stalling logic to the pipeline.

To start, we need to add the commands `LOAD` and `STORE` to the `Cmd` type:

```
data Cmd = ADD | SUB | INC | LOAD | STORE
```

We also need to define some additional Hawk circuits. The first circuit, `kill`, takes a *kill* signal and a signal of transactions. On each clock cycle, the `kill` component returns its transaction input unchanged, unless the kill signal is asserted, in which case it returns the `nop` transaction:

```
kill :: Signal Bool -> Signal Transaction -> Signal Transaction
kill ks inp = select ks (constant nop) inp
```

The `isLoadTrans` circuit returns `True` whenever its argument signal is a load transaction:

```
isLoadTrans :: Signal Transaction -> Signal Bool
isLoadTrans ts = lift isLoad ts
  where
    isLoad (Trans _ cmd _ _) = (cmd == LOAD)
```

Although we previously passed SHAM instructions as parameters, we now need to call a function, `instrCache`, to explicitly retrieve them:

```
instrCache :: Signal Bool -> Signal Transaction
```

Since the pipeline can stall, we need a way to ask for the same instruction two cycles in a row. The `instrCache` function takes a Boolean signal and returns the current transaction. Whenever the argument signal is `True`, then on the next cycle `instrCache` returns the same transaction as it did for the current clock cycle. Otherwise, it returns the next transaction as normal.

We also need a circuit that actually performs the loads and stores:

```
mem :: Signal Transaction -> Signal Transaction
```

On those clock cycles where the input transaction is anything but a load or store transaction, the `mem` function simply returns the transaction unchanged. On loads, `mem` updates the destination operand of the input transaction, based on the input load address. On stores, `mem` updates its internal memory array according to the address and contents given in the input transaction. The destination operand value is set to zero.

We also define a new Hawk function, `transHazard`, that returns `True` whenever its two transaction arguments would cause a hazard, if the first transaction preceded the second transaction in a pipeline:

```
transHazard :: Signal Transaction -> Signal Transaction -> Signal Bool
```

The extended Hawk specification using transactions is given below:

```
SHAM3Trans :: Signal Transaction
SHAM3Trans = memOut'
  where
     -- register-fetch stage --
     instr       = instrCache loadHzd
     readyInstr  = regFile instr memOut'
     readyInstr' = delay nop (kill loadHzd readyInstr)

     -- ALU stage --
     aluIn   = bypass (bypass readyInstr' memOut') aluOut'
     aluOut  = alu aluIn
     aluOut' = delay nop aluOut

     -- memory stage --
     memIn   = bypass aluOut' memOut'
     memOut  = mem memIn
     memOut' = delay nop memOut

  ----- Control logic -----
     loadHzd = sigAnd (isLoadTrans readyInstr')
                      (transHazard readyInstr' readyInstr)
```

The register-fetch stage retrieves the instruction and fills in its source operands from the register file. The register-fetch pipeline register delays the transaction by one clock cycle, although if there is a load hazard, the register instead outputs a nop transaction on the next cycle. The ALU stage first updates the source operands of the stored transaction with the results of the two preceding transactions (`memOut`' and `aluOut`') by invoking bypass twice. It then performs the corresponding ALU operation, if any, on the transaction and stores it in the ALU-stage pipeline register. The memory stage again updates the stored transaction with the immediately preceding transaction, performs any required memory operation, and stores the transaction. The stored transaction is written back to the register file on the next clock cycle. The control logic section determines whether a load hazard exists for the current transaction, that is, whether the immediately preceding transaction was a load instruction that is in hazard with the current transaction.

As we can see, the body of the specification remains manageable. The small control logic section to detect load hazards is straightforward and is a minority of the overall specification. In contrast, an equivalent specification of this pipeline where the components of each transaction were explicitly represented contained over three times as many source lines. The lower-level specification's control section was almost as large as the dataflow section, and not nearly as intuitive.

2.2.10 Extending transactions to other microarchitectures

The essential idea behind transactions is to pass all of the microarchitectural state associated with a particular instruction in a single data structure as the instruction traverses the pipeline. This implies that more transaction fields may have to be created for more sophisticated pipelines. For example, the pipelined microarchitecture of Chapter 3 performs *branch speculation*, where the instruction fetching component predicts the address of the next instruction to be executed after a branch, called the *branch target*. This allows the pipeline to continue fetching and executing instructions even though the actual branch target won't be known until the ALU component has computed it in a later pipeline stage. If the prediction is incorrect, the pipeline must discard the transactions corresponding to instructions it had fetched after the branch, and start fetching the correct branch successor instructions instead.

For pipelines containing branch speculation the predicted branch target is part of the microarchitectural state associated with the branch, and is therefore stored within the branch's transaction structure. This turns out to be quite useful when the actual branch target is calculated by the ALU component, since it can be compared to the speculated branch target to determine if a misprediction occurred.

Other microarchitectural features such as virtual register tags, predication bits, exception status flags, etc. may also require modifications to the transaction type. Haskell's type class mechanism can be used to create structured families of transaction types that can be instantiated to particular microarchitectures, depending on what state needs to be associated with a given instruction. The use of type classes in Hawk to abstract over microarchitecture features is presented in Cook et al[18]. We do not follow this approach in the thesis, however, since we will be dealing with a fixed set of microarchitectural features.

2.2.11 Transactions in other modeling languages

We are not alone in noting the usefulness of transactions to regularize interfaces between microarchitecture components. In particular, Önder and Gupta have used a similar concept of *instruction contexts* as a core datatype in UPFAST, an imperative microarchitecture simulation language [72]. Instruction contexts are allocated as mutable records as instructions are fetched. They are then passed along by *components*, which can imperatively update context fields, if desired. A context is deallocated when it is no longer needed.

Transactions have also been used by others to structure microarchitecture verifications, and their use for this purpose is discussed in Section 3.3.1.

2.3 Modeling the DLX

Using techniques comparable to those described in this chapter the Hawk team has modeled several DLX architectures:

- An unpipelined version, where each instruction executes in one cycle.
- A pipelined version where branches cause a one-cycle pipeline stall.

- A more complex pipelined version with branch prediction and speculative execution. Branches are predicted using a one-level branch target buffer. Whenever the guess is correct, the branch instruction incurs no pipeline stalls. If the guess is incorrect, the pipeline stalls for two cycles.

- An out-of-order, superscalar microprocessor with speculative execution. The microarchitecture contains a reorder buffer, register alias table, reservation station, and multiple execution units. Mispredicted branches cause speculated instructions to be aborted, with execution resuming at the correct branch successor. Cook et al[18] present an overview of this microarchitecture and its implementation in Hawk.

The microarchitectural specification for the unpipelined DLX is written in a quarter page of uncommented source code, not including the reusable component definitions; the most complicated pipelined version takes up just over half a page.

2.3.1 Executing the model

We used the Gnu C compiler that generates DLX assembly to test our specifications on several programs[2]. These test cases include a program that calculates the greatest common divisor of two integers, and a recursive procedure that solves the towers of Hanoi puzzle.

We have not made detailed simulation performance measurements on these pipeline specifications. In general we do not expect the current implementation of Hawk to break simulation-speed records. At the moment Hawk is a set of libraries written in a general-purpose lazy functional language, which imposes some performance costs. The transaction library also performs some run-time tests that would be "compiled-away" in a lower-level pipeline specification. We hope to increase Hawk's simulation efficiency in the future by investigating domain-specific compilation techniques, such as partially evaluating a Hawk microarchitecture with respect to the program it is simulating and the output signals being sampled. Performance could also be greatly improved by employing custom memory

[2]Thanks are due to Byron Cook for developing the DLX assembly to Hawk translator, and for integrating it with the Gnu C compiler

allocation algorithms that take into account the fact that most Hawk programs only reference a small "window" of a signal at any given clock cycle.

2.4 Other hardware modeling languages

Currently the hardware modeling language that is the most similar to Hawk is Lava, introduced in Section 2.1.5. However, a major point of departure from Hawk is Lava's ability to treat signal *descriptions* as first class values. In Hawk, a signal is simply a sequence of values, and there is no way to differentiate between two signal descriptions that happen to generate the same sequence. For example, the following two Hawk circuits are observationally equivalent:

```
toggle :: Signal Bool
toggle = delay False (sigNot toggle)

toggle' :: Signal Bool
toggle' = genDelays False
  where
    genDelays :: Bool -> Signal Bool
    genDelays b = delay b (genDelays (not b))
```

The first Hawk definition, `toggle`, describes a simple toggling circuit implemented by a feedback loop (`sigNot` is an inverter over boolean signals). A circuit could be naturally synthesized from this description using a single inverter and delay component. The second definition, `toggle'`, makes use of the recursively-defined `genDelays` function to describe an infinite number of delay components with alternating initial value parameters. Each delay component takes the rest of its values from the next delay component to be generated. The `toggle'` circuit description is not realizable in hardware, yet both `toggle` and `toggle'` generate the same sequence of values `<False, True, False, ...>` and are therefore equal in Hawk.

Lava can detect that these two circuit descriptions are different. Lava accomplishes this by extending the Haskell language slightly with a form of *non-updatable reference*[13]. Lava references act much like ordinary references in impure functional languages such as ML, except that they are "read-only". Once initialized, a Lava reference cannot be modified. Lava references differ from applicative data structures in that a newly created reference is distinct from any already existing reference, even if they both refer to the same value. Lava has an equality operator on references that can test whether its two reference parameters are in fact the same reference.

A Lava signal is then a reference to the Lava component whose output generates the signal. A Lava component is a record containing a field for the component's name, such as `"delay"` or `"not"`, a field for each static component parameter, such as the initial value parameter for a `delay` component, and a field for each of the the component's input signals. By performing equality tests on Lava signals, a Lava program can distinguish `toggle` from `toggle'`, since the first circuit generates only two unique references, one each for the `delay` and `sigNot` components, while the second circuit (lazily) generates an unbounded number of references.

Given a Lava description of a circuit, one can write a Lava function that generates its corresponding behavior as an infinite list. Lava also allows users to generate *non-standard interpretations* of circuits such as netlists and state-machine descriptions. Generation of non-standard interpretations is a powerful Lava capability. Lava has circuit interpretations that

- synthesize VHDL code.
- generate circuit formulas that can be checked by several verification tools, such as Gandalf[89], NP-tools[78], and Otter[56].

Unfortunately, Lava's ability to generate non-standard circuit interpretations comes at the price of giving up pattern-matching over signal elements. Haskell currently has a fixed interpretation of pattern-matching expressions which is incompatible with Lava's explicit signal representations. For example, it is quite convenient in Hawk to define an instruction opcode as an algebraic datatype (see Section 7.1.1) and then define components such as the ALU in terms of functions that pattern match on the opcode constructors. While the same functionality can be defined in Lava by bundling existing signal types and performing tests through nested conditional expressions[3], the resulting code is often more verbose and less easy to read.

Given the current state of the Haskell language, one has to choose between being able to define non-standard circuit interpretations versus defining signal transformers by pattern matching. Each has significant advantages. Since Hawk is primarily a behavioral specification language, we chose the latter.

Custom-designed languages

Of course, Haskell is not the only possible platform for designing hardware description languages. Most, but not all, hardware modeling languages are designed "from scratch", giving designers complete control, and responsibility, over the syntax, semantics, and tooling infrastructure of the language.

For example, Daisy[36] and μFP[38] are examples of early hardware specification languages based on higher order functional languages. Daisy as originally developed in Johnson's dissertation is a lazy untyped functional language where circuits are specified as recursive signal equations, as they are in Hawk. The semantics of recursive definitions is given in terms of domain-theoretic least fixed points, rather than unique fixed points as used in this dissertation (unique fixed points are introduced in Section 4.5). Domain theoretic semantics are arguably more complex to reason about in a theorem prover than Hawk's higher order logic semantics, but have the advantage of allowing circuit equivalences to be proved directly via an elegant technique called *fixpoint induction*.

μFP is a combinator-based language. Whereas Daisy and Hawk allow arbitrary recursive signal forms, in μFP all recursion is expressed through a set of higher order recursive combinator functions. In addition, μFP circuit components are connected via function composition, without explicitly naming the interface signals. Two advantages of such *point-free* specification languages are the simplicity of the language and supporting tools,

[3]Lava also has the ability to define new *abstract* signal types, with an associated set of abstract signal primitives. Each circuit interpretation must provide a definition of the primitives if it is to interpret circuits containing the abstract signal type.

and the ability to specify *layout* directives. μFP's layout combinators allow circuit designers to state in high-level terms where circuit components should be realized on silicon. These directives are more difficult to implement in languages like Hawk that allow components to be interconnected arbitrarily.

The Ruby[39] hardware description language is a successor to μFP. Created by Jones and Sheeran, Ruby is a combinator language based on relations, rather than functions. Circuit specifications in Ruby can be more general than in Hawk, in that relations can describe more circuits than functions can. For example, a Ruby circuit can directly model a *bi-directional* wire between two circuits C and D, such as a bus, where information flows from C to D on some clock cycles, and from D to C on others. Hawk's functional basis requires all wires to be uni-directional. A bi-directional wire between circuits C and D in Hawk must be modeled as two signals, one signal returned as an output from C and passed as an input to D, and the other returned from D and passed into C.

Ruby can also model a *nondeterministic* circuit, whose outputs are not uniquely determined by its inputs. Ruby's support for nondeterminism enables a form of *design by refinement*, which we discuss in Section 7.1.5.

Most of the published Ruby examples specify circuits that operate at the gate and word level, and particularly circuits that contain fine-grained regular structure, such as systolic arrays. Such circuits generally process collections of fairly simple forms of data, such as vectors of booleans and numbers. Hawk has emphasized modeling the more complex, but less regular datatypes that typify microarchitecture component interfaces. Thus Hawk programs can declare algebraic datatypes and define circuits by pattern-matching, features which Ruby lacks.

Ruby's emphasis on circuit layout is another example of the different set of design goals between the two languages. Ruby has combinators to specify where circuits are located in relation to each other and to external wires. Hawk's emphasis is on behavioral correctness, so Hawk circuits do not contain layout information.

There are many other languages for specifying hardware circuits at varying levels of abstraction. The most widely used such languages are Verilog and VHDL. Both of these languages are well suited for their roles as general-purpose, large-scale hardware design languages with fine-grained control over many circuit properties. Both of these languages are more general than Hawk in that they can model asynchronous as well as synchronous circuits, and can synthesize (a subset of) circuit descriptions into a form suitable for fabricating in silicon. However, Verilog and VHDL are large languages with complex event-simulation semantics, which makes circuit verification much more difficult (see, for example, Gordon[30] for the challenges in formally verifying Verilog circuits). Also, neither of these languages supports higher level abstraction features as well as Hawk, such as polymorphically-typed circuits and higher-order circuit combinators.

As part of Intel Corporation's *Forte* circuit verification environment, the lazy functional language Lifted-FL[2, 3] is used as a meta-language for describing abstract circuit models, circuit properties, and circuit verification algorithms. Lifted-FL extends the `bool` datatype to contain symbolic boolean expressions, which are represented as ordered binary decision diagrams[17] (BDDs). Synthesized gate-level circuit descriptions can be imported as Lifted-FL data structures from several conventional net-list file formats. Once imported, the circuits can be symbolically simulated and verified. The simulation and verification algorithms are written in Lifted-FL at a high level of abstraction, due in part to the language's support for higher order functions and algebraic datatypes, but also due to its intrinsic support for symbolic boolean expressions. Lifted-FL has been used to verify impressively large circuits, including several floating-point ALU cores[71].

HML[48, 49] is a hardware modeling language based on the functional language ML. ML also has higher-order functions and static polymorphic type checking, allowing many of the same abstraction techniques that are used in Hawk, with similar safety guarantees. HML follows the tradition of VHDL and Verilog in expressing circuit modules in a relational style, where output signals become extra parameters, rather than returned values as in Hawk. The goal of HML is also rather different from Hawk, concentrating on circuits that can be immediately realized by translation to VHDL.

Chapter 3

Microarchitecture algebra

3.1 Introduction

We now turn from specifying and simulating microarchitectures written in Hawk to developing a method for verifying them. This thesis approaches the verification task algebraically, by discovering behavior-preserving transformations for Hawk components. Transformational laws are well known in digital hardware, and form the basis of logic simplification and minimization, and of many retiming algorithms. Traditionally, these laws occur the gate level: de Morgan's law being a classic example. In this chapter we examine whether corresponding transformational laws hold at the microarchitectural level.

A priori, there is no reason to think that large microarchitectural components should satisfy any interesting algebraic laws, as they are constructed from thousands of individual gates. Boundary cases could easily remove any uniformity that has to exist for simple laws to be present. Yet we have found that when microarchitectural units are presented as transaction processors, many powerful laws appear. Moreover, as we demonstrate in this chapter, these laws *by themselves* are powerful enough to allow us to show equivalence of pipelined and non-pipelined microarchitectures.

We have used this algebraic approach to simplify a pipelined microarchitecture that uses forwarding, branch speculation and pipeline stalling for hazards. The resulting pipeline is very similar to the reference machine specification (i.e. no forwarding logic), while still retaining cycle-accurate behavior with the original implementation pipeline. The top-level transformation proof is simple enough to be carried out on paper, and can also be automated to some extent using Isabelle's higher-order rewriting tactics.

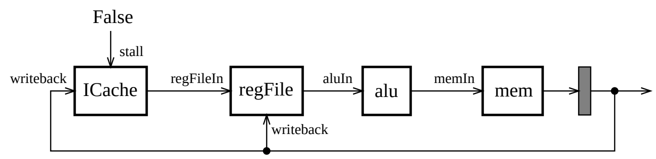


Figure 3.1: One-stage pipeline.

Interestingly, both circuits and laws can be expressed diagrammatically. A paper proof (transformation using equivalence laws) proceeds as a series of microarchitecture block diagrams, each an incrementally transformed version of the last. The laws often have a geometric flavor to them, such as laws to swap two components with each other, or laws to absorb one component into another. We find this diagrammatic approach an excellent way to communicate proofs.

The most time-consuming part of this technique has been discovering the local behavior-preserving laws. It is our experience that these laws are much easier to discover when using transactions to increase the level of abstraction. Not only do transactions reduce the size of microarchitecture specifications, they also provide enough "auxiliary" state information to make law-discovery practical.

The rest of the chapter discusses many of the laws we have discovered. We then show their use by applying the laws in a proof of equivalence between two microarchitectures.

3.2 Reference microarchitecture

Figure 3.1 shows the diagram of a simple non-pipelined microarchitecture built out of transaction signal processors. The components are the same as those used in the `SHAM3Trans` microarchitecture in Sections 2.2.5 and 2.2.9, but have been augmented to handle branch instructions. In particular, the `alu` component computes target addresses for branch transactions, and the `iCache` examines completed branch transactions to determine when to change its internal PC. The textual Hawk description is shown in Figure 3.2: Like its `SHAM3Trans` counterpart, the `iCache` component produces new transactions, based on the value of the current program counter and the contents of program memory (the instruction-set architectures we consider have separate address spaces for instructions and

```
referenceMA = writeback
where
  regFileIn = iCache (constant False) writeback
  aluIn     = regFile regFileIn writeback
  memIn     = alu aluIn
  memOut    = mem memIn
  writeback = delay nop memOut
```

Figure 3.2: Hawk code for reference microarchitecture

data). Both the current PC and the instruction memory contents are internal to `iCache`. The instruction cache takes on its `writeback` input the completed transaction from the previous clock cycle. It examines each writeback transaction for branches that have been taken. When it finds such an instruction, it modifies its internal PC accordingly and starts fetching transactions from the branch target address. The iCache has as output a signal of transactions representing the newly-fetched instructions. Each transaction's source and destination operand values are initialized to zero, since the iCache doesn't know what values they should have[1]. The other pipeline components will fill in these fields with their correct values. The iCache has a second input, called `stall`, which is a signal of Boolean values. On clock cycles where `stall` is asserted, the iCache will output the same transaction as it did on the previous clock cycle. In this simple microarchitecture, `stall` is always false. In more complex pipelines, the `stall` signal is typically asserted when the pipeline needs to stall due to a branch misprediction.

For more complex pipelines, we also allow the iCache to perform branch prediction, based on an internal branch target buffer. When performing branch prediction, the iCache will also annotate branch instruction transactions with the predicted branch target PC. A `branch_misp` component (not shown in Figure 3.1) can locally compare the predicted branch target with the actual branch target to determine if a branch misprediction has occurred. For branch predicting microarchitectures the iCache updates its internal PC on all mispredicted branches, once they are received on the writeback input, rather than on

[1] The `SHAM3Trans` version of the `iCache` component returned `Unknown` for the uninitialized operand values. This version of `iCache` will instead simply zero out the operand value fields, to simplify the proofs given in Chapter 6

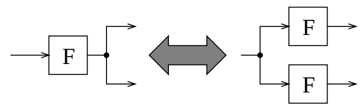


Figure 3.3: Universal circuit-duplication law

taken branches.

3.3 Algebraic reasoning and the microarchitecture laws

With any algebraic reasoning there need to be some ground rules. We take as fundamental the notion of *referential transparency* or, in hardware terms, a *circuit duplication law*. Any circuit whose output is used in multiple places is equivalent to duplicating the circuit itself, and using each output once. This law is shown graphically in Figure 3.3. Because of the declarative nature of our specification language, every circuit satisfies this law. That is, it is impossible within Hawk for a specification of a component to cause hidden side-effects observable to any other component specification. In many specification languages this law does not hold universally. For example, duplicating a circuit that incremented a global variable on every clock cycle would cause the global variable to be incremented multiple times per clock period, breaking behavioral equivalence. Hawk circuits can still be stateful, but all stateful behavior must be local and/or expressed using feedback.

3.3.1 Algebraic reasoning

Referential transparency is what allows us to use algebraic reasoning effectively in Hawk, and is based on the referentially-transparent semantics of Haskell. In general, algebraic techniques for transforming functional programs are routinely used for equivalence checking and verification [7, 8, 43] and for compilation and optimization [26, 77]. Much of the work in this thesis can be seen as an extension of these ideas.

We have also been influenced by the algebraic techniques used in the relational hardware-description language Ruby[84] (Ruby is described in Section 2.4). Sizeable Ruby circuits have been successfully derived and verified through algebraic manipulation[37, 40], and a formal semantics of a dependently-typed subset of Ruby, called T-Ruby, has been mechanized within Isabelle's Zermelo-Fraenkel set theory logic[79]. On top of the formal semantics, the T-Ruby design system[85] has been built as a set of tools to algebraically transform Ruby expressions and translate hardware-realizable T-Ruby circuits into structural VHDL. The rewrite rules are verified within Isabelle's theory of T-Ruby circuits.

What distinguishes our work is our focus on microarchitectural units as objects of study in their own right, whereas the Ruby research has emphasized circuits at the gate level. Hawk's model of time is also somewhat different than Ruby's. Hawk uses natural numbers as time indexes, while Ruby uses integers. One place where this difference shows up is the fact that Ruby *delay* components form a bijection on signals, while Hawk *delay* components do not (they are injective, however). The bijectiveness of Ruby's *delay* components make it somewhat simpler to retime circuits in that language. Another important difference is Hawk's greater emphasis on proving circuits equivalent by performing induction over time, as occurs in Chapter 6. Ruby's integer-indexed signals do not permit this form of reasoning.

Transactions

Transactions are a key concept in allowing us to discover and formulate many of the algebraic laws of microarchitectural components. As we noted in Chapter 2, the usefulness of transactions for verification has been noticed before. Here we observe their uses in verification. For example, Aagaard and Leeser used transactions to specify and verify hierarchical networks of pipelines[1]. Further, Sawada and Hunt use an extended form of transactions in their verification of a speculative out-of-order microarchitecture [82]. Each transaction records two snapshots of the entire ISA state, before and after the instruction is executed. In their work, however, transactions are not part of the microarchitecture itself, but are constructed separately for verification purposes.

In our work, transactions form a fundamental basis for algebraic laws over microarchitectural components. The next few sections introduce many such laws, some of which are specific to particular combinations of components, while others are quite widely applicable. Each instantiation of a law needs to be proved with respect to the specification

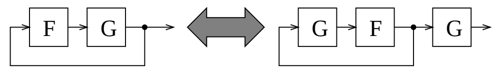


Figure 3.4: feedback rotation law

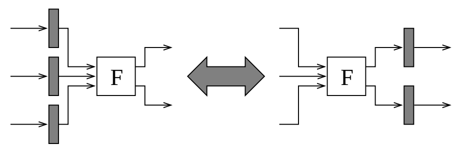


Figure 3.5: time-invariance law.

of the circuit components involved. We do not verify the individual laws in this chapter, but several are proved correct using induction and equational reasoning in Chapter 6.

3.3.2 Delay laws

The delay circuit is a fundamental building block of clocked circuits, especially when combined with feedback. A feedback variant of the circuit duplication law shown in Figure 3.4, called the *feedback rotation* law, allows circuits to be split along feedback wires. This law is not universal, but it is valid for any circuit that does not contain zero-delay cycles.

Happily, all of the laws we discuss, including the feedback rotation law itself, preserve a well-formedness property: if a circuit contains no zero-delay cycles, then any transformed circuit will also have no zero-delay cycles.

The *time-invariance* law (Figure 3.5) is also widely applicable. A circuit is *time-invariant* if one can retime the circuit by removing the `delay`s from all the inputs of the circuit and placing new `delay`s (with possibly different initial value parameters) on the circuit's outputs. All combinatorial circuits are time-invariant, and so are many stateful circuits like the register file and memory cache. Interestingly, the iCache is not as it can track the passage of time since initialization.

We use the above laws extensively to remove pipeline stages. If a pipeline stage is time-invariant, then we can move the pipeline registers (represented as `delay` circuits)

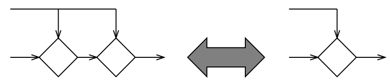


Figure 3.6: bypass circuit idempotence law

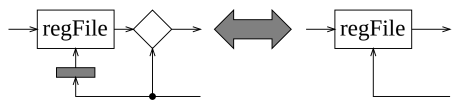


Figure 3.7: register-bypass law

from before the pipeline stage to afterwards. If subsequent pipeline stages are also time-invariant then we can repeat the process, eventually moving all of the delay circuits to the end of the pipeline. However, forwarding logic between pipeline stages must still access the appropriate time-delayed outputs of later pipeline stages. The feedback-rotation law polices this, and ensures that the appropriate time-delay is kept by forcing `delay`s to be inserted on all feedback wires to the forwarding circuits. We will see examples of this enforcement in Section 3.4.

The movement of delay components is an application of a technique called *retiming*[45, 83, 86]. A circuit is retimed when the delay components of the circuit are repositioned, while the functional components are left unchanged, Typically, circuits are retimed to reduce the clock cycle time. In contrast, we shall retime circuits as part of a simplification process. In fact, we often use the time invariance law to increase cycle time!

3.3.3 Bypasses and bypass laws

The purpose of bypass components as defined in Section 2.2.7 is to ensure that results computed in later pipeline stages are available to earlier pipeline stages in time to be used. Bypass circuits have many nice properties. Not only are they time-invariant and obey a kind of idempotence (Figure 3.6), but they also interact closely with register files and various execution units.

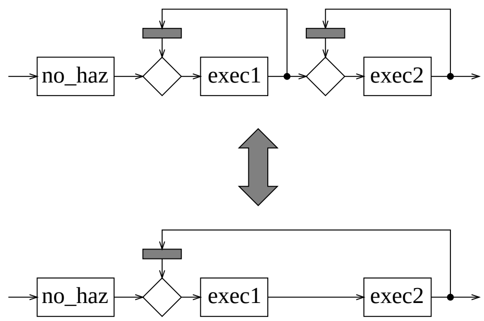


Figure 3.8: hazard-bypass law

Register file - bypass law

The fundamental interaction between a bypass and register file is shown in Figure 3.7. We call this the *register-bypass law*, and it is used repeatedly in eliminating forwarding logic when simplifying pipelines. The law states that we can delay writing a value into the register file, so long as we also take the value to be written and forward it to the output, in case that register was being read on the same clock cycle.

Hazard - bypass law

Another bypass law permits the removal of bypasses between execution units. It is often the case that after retiming all delay circuits to the end of a pipeline, two execution units in a pipeline (such as an ALU unit and a Load/Store unit) are connected with one-cycle feedback loops. Each bypass circuit is forwarding the outputs of an execution unit to the inputs of that same execution unit, one clock cycle later.

If the upstream pipeline stages can guarantee that there is no hazard between successive transactions, then the double feedback is equivalent to the single feedback circuit shown at the bottom of Figure 3.8. This (conditional) identity is called the *hazard-bypass* law.

To be more concrete, suppose exec1 is the ALU and exec2 the memory cache. Then an ALU-mem hazard arises if a transaction which loads a register value from memory is immediately followed by an ALU operation which requires that register's value (this is the same hazard as the one presented in Section 2.2.9). Under these circumstances the two feedback loops would give different results. Under all other circumstances the two circuits

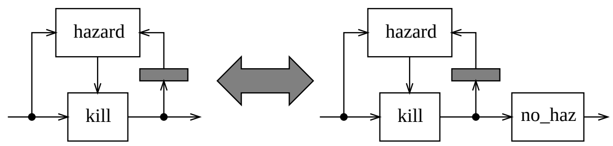


Figure 3.9: Hazard-squashing logic guarantees no hazards

are equivalent. We express this conditional equivalence using the `no_haz` component. It is an example of a projection component and is discussed in the next section.

3.3.4 Projection laws

Many laws, like the hazard-bypass law above, require that the input signals satisfy certain properties, and commonly, we may know that the output signal of a given component always satisfies a particular property. We can capture this knowledge of properties using signal *projections*.

A signal projection is a component with one input and one output. As long as the input signal satisfies the property of interest, the component acts like an identity function, returning the input signal unchanged. However, if the input does not satisfy the property we are interested in, the projection component modifies the input signal in some arbitrary way so that the property is satisfied.

Let us consider an example. For the hazard-bypass law we are interested in expressing the absence of ALU-mem hazards in a transaction signal. We reify this property as a `no_haz` projection. On each clock cycle, the `no_haz` component compares the current input transaction with the previous input transaction. If there is no ALU-mem hazard between the two transactions, then the current transaction is output unchanged. If a hazard does exist, then `no_haz` will instead output `nopTrans`, which is guaranteed not to generate a hazard (since `nopTrans` contains no source operands).

Where do projections come from? After all, they are not the sort of component that microarchitectural designers introduce in the normal course of events.

Fig 3.9 provides an example of a law which “generates” a projection. The hazard-squashing logic guarantees that its output contains no hazards, and this is expressed in that the circuit is unchanged when the no_haz component is inserted on its output.

(The hazard component outputs a Boolean on each clock cycle stating whether its two input transactions constitute a hazard. The kill component takes a transaction signal and a Boolean signal as inputs. On each clock cycle, if the Boolean input is false, then kill outputs its input transaction unchanged. If the Boolean input is true, then kill outputs a nopTrans, effectively "killing" the input transaction.)

To be useful, a projection component needs to be able to migrate from a source circuit that produces it (such as the circuit in Figure 3.9) to a target circuit that needs the projection to enable an algebraic law (such as the hazard-bypass law). Thus a projection component must be able to commute with the intervening circuits between the source and the target circuit. Well-designed projections commute with many circuits. For instance, the no_haz projection commutes with bypass, alu, mem, and regFile components. It also commutes with delay components (that is, no_haz is time-invariant).

Projections are also convenient for expressing the fact that a component only uses some of the fields of an input transaction. For instance, the hazard component only looks at the opcode, source, and destination register name fields of its two input transactions. We can create a projection called proj_ctrl that sets every other field of a transaction to a default value, and prove a law stating that the hazard component is unchanged when proj_ctrl is added to any of its inputs. We can then show that proj_ctrl commutes with other components, such as bypasses and delays. This allows us to move the input wires to hazard across these other components, which is sometimes necessary to enable other laws. Similarly, the proj_branch_info projection allows us to move iCache and branch_misp component inputs.

3.4 Transforming the microarchitecture

The laws we have been discussing can be used for aggressively restructuring microarchitectures while retaining equivalence. We have used them to simplify several pipelined microarchitectures with a view to verification. The example we present here contains three levels of forwarding logic, resolves hazards by stalling the pipeline, and performs

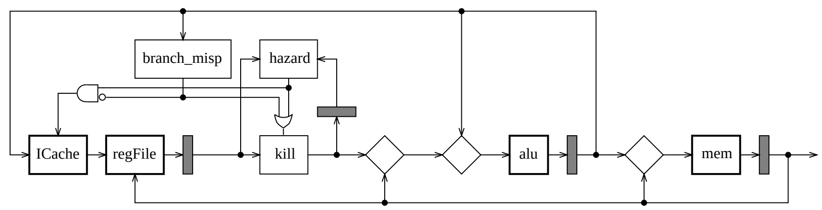


Figure 3.10: Microarchitecture before simplification

branch speculation. The block diagram for this microarchitecture is shown in Figure 3.10. By using just algebraic laws, we have been able to reduce most of the complexity, leaving essentially an unpipelined microarchitecture.

Our approach to pipeline simplification has echoes of the *Unpipelining* approach[46] of Levitt and Olukotun. Unpipelining is a verification technique where a pipelined microarchitecture, specified as a state machine, is incrementally transformed into a functionally-equivalent unpipelined microarchitecture. Unpipelining proceeds by repeatedly merging the last stage of a pipeline into the next to last stage, producing a microarchitecture with one less stage on each iteration. On each iteration, the two microarchitectures are proven equivalent by induction over time. This is similar to our approach, except that we use transactions to encapsulate and reuse many of the verification steps, and we only need to prove the equivalence of the portion of the microarchitecture being transformed, rather than the entire microarchitecture, on each iteration. On the other hand, Levitt and Olukotun's implementation of unpipelining is much more automated than our work up to now, and can completely reduce a pipelined implementation to an unpipelined reference machine.

The simplification of the microarchitecture in Figure 3.10 proceeds in five goal-directed stages: Retiming, moving control wires, propagating hazard information, removing forwarding logic, and cleanup. The stages are chosen somewhat arbitrarily, and are fairly specific to this microarchitecture. They nevertheless help to organize the top-level proof into subgoals. Each stage is described as we come to it in the simplification, and achieves the preconditions necessary to apply key microarchitecture laws in the next stage. The retiming stage is described next.

3.4.1 Retiming stage

We first remove all delay circuits from the main pipeline path, starting at the earliest stage in the pipeline. We accomplish this by repeatedly applying the time-invariance law, and by splitting delays along wires through the circuit duplication and feedback rotation laws.

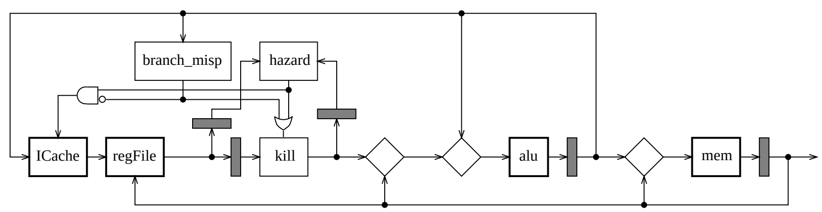


Figure 3.11: Split `delay` circuit after `regFile`, using the circuit duplication law

We would now like to move a `delay` through the `kill` circuit, but we can't, since the top input to `kill` does not have a `delay` circuit. To place a `delay` on `kill`'s top input, we will need to move `delay` circuits through the `branch_misp` and `hazard` circuits. This is possible because `branch_misp` and `hazard` are pure combinational circuits that preserve default values (The default value for Booleans is `False`) and are therefore time-invariant.

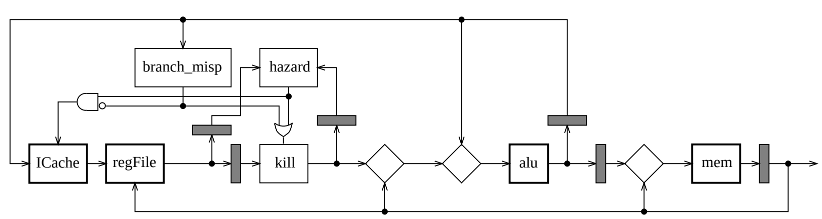


Figure 3.12: Split `delay` circuit after `alu`, using the feedback-rotation law

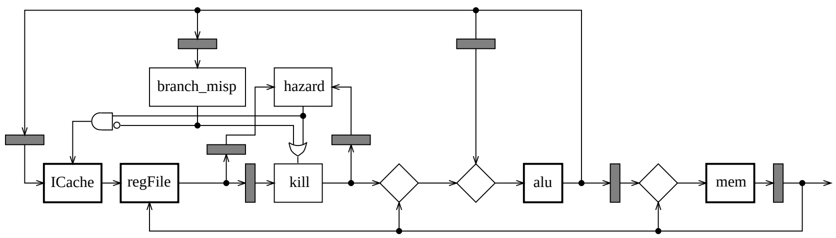


Figure 3.13: Split twice the `delay` circuit leading to `branch_misp` and `iCache`, using two applications of the circuit-duplication law

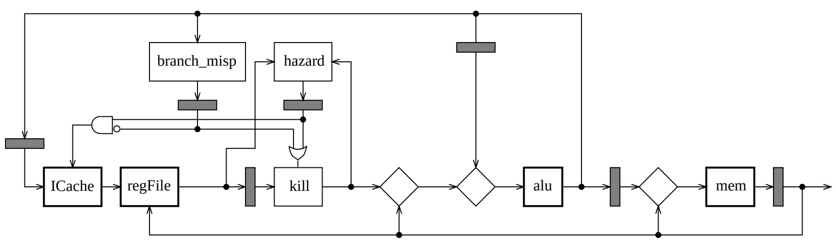


Figure 3.14: Move `delay` circuits through the `branch_misp` and `hazard` circuits, using the corresponding time-invariance laws

We can similarly move these `delay` circuits through the `or` and `and` circuits (even though one of the `and` inputs is inverted), since these combinational circuits preserve the default `False` Boolean value. Finally, we can move the original `delay` circuit through the `kill` circuit, since `kill` is a combinational circuit and all of its inputs have delays.

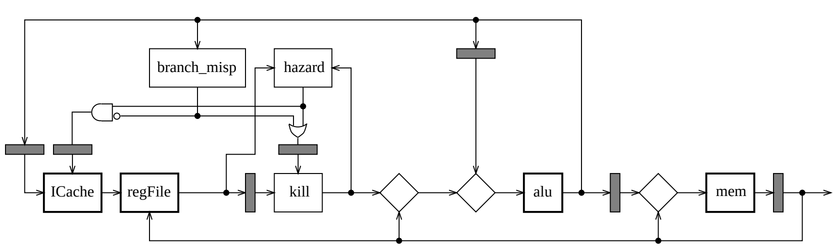


Figure 3.15: Move `delay` circuits through the `or` and `and` circuits, using the circuit-duplication law and the corresponding time-invariance laws

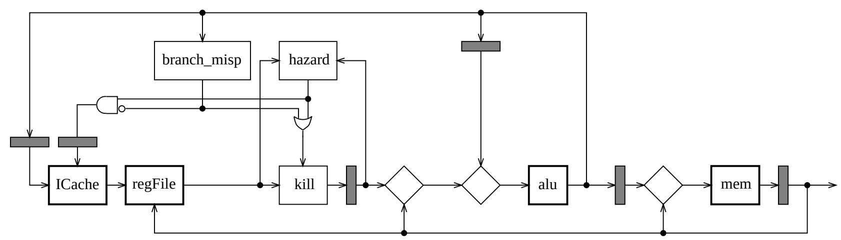


Figure 3.16: Move `delay` circuits through the `kill` circuit, using the corresponding time-invariance laws

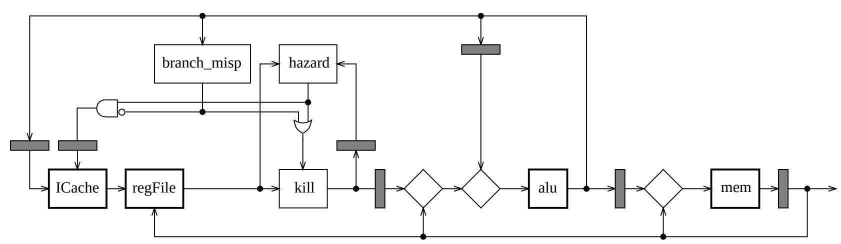


Figure 3.17: Split the `delay` circuit after the `kill` circuit, using the circuit duplication law

Once again, we can't move the `delay` circuit past the `bypass` circuit, since the other input to the bypass does not contain a `delay`. Fortunately, the other input originates at the `delay` circuit that is after the `mem` circuit, so we split that `delay` and move it to the `bypass` input.

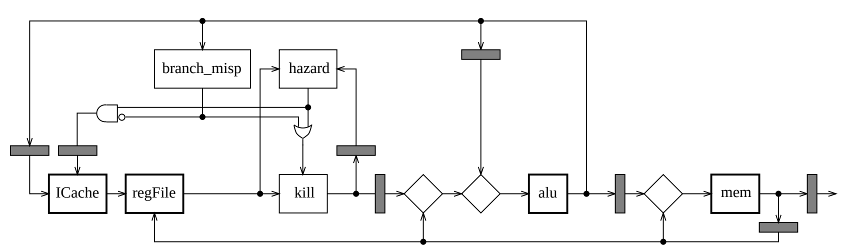


Figure 3.18: Split the `delay` circuit after the `mem` circuit, using the feedback rotation law

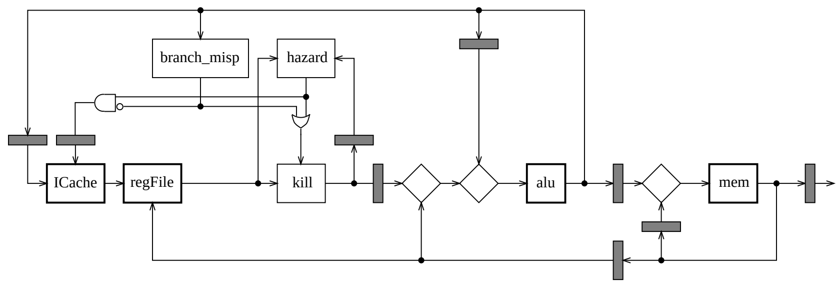


Figure 3.19: Split the bottom-most `delay` circuit, using the circuit duplication law

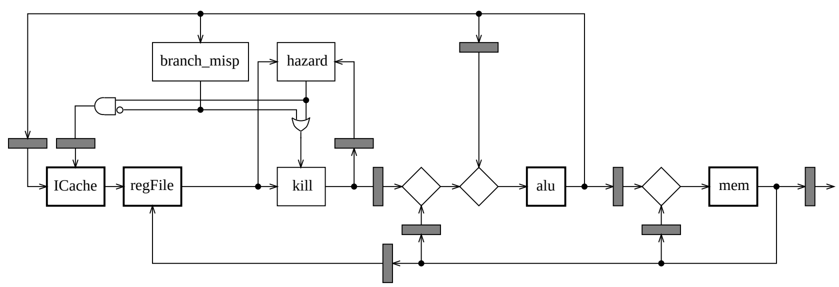


Figure 3.20: Split the bottom-most `delay` circuit again, using the circuit duplication law

We can now move our wandering `delay` through the two `bypass` circuits, since bypasses are time-invariant, and they both have `delay` circuits on all inputs.

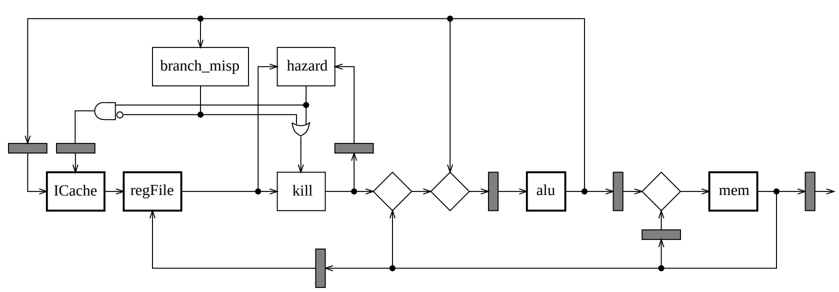


Figure 3.21: Move the `delay` circuit before the first `bypass` circuit through the first and second bypasses, using the corresponding time-invariance laws

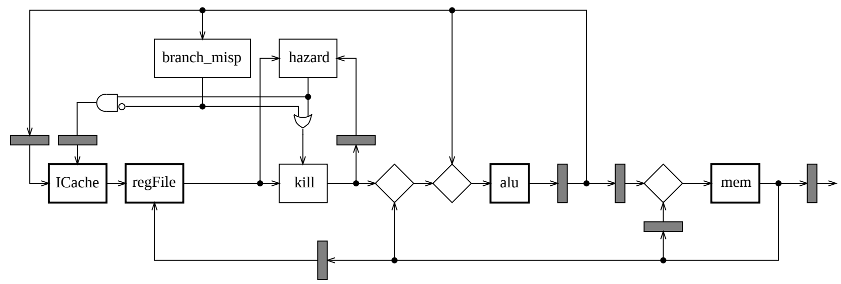


Figure 3.22: Move the `delay` circuit through the `alu` circuit using the corresponding time-invariance law

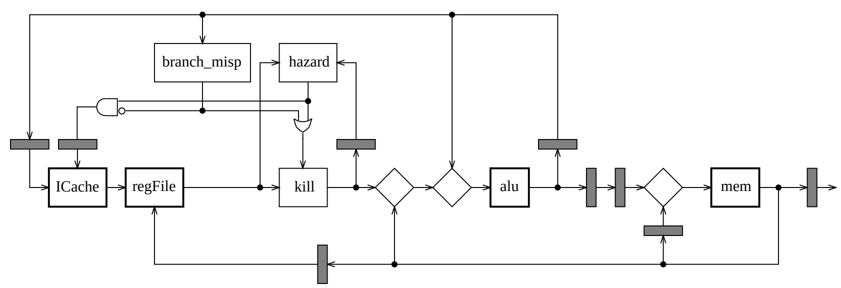


Figure 3.23: Split the `delay` circuit after the `alu` circuit using the feedback-rotation law

Now we just have to move the two `delay` circuits before the third `bypass` circuit to the end of the pipeline. Fortunately, both `bypass` and `mem` are time-invariant.

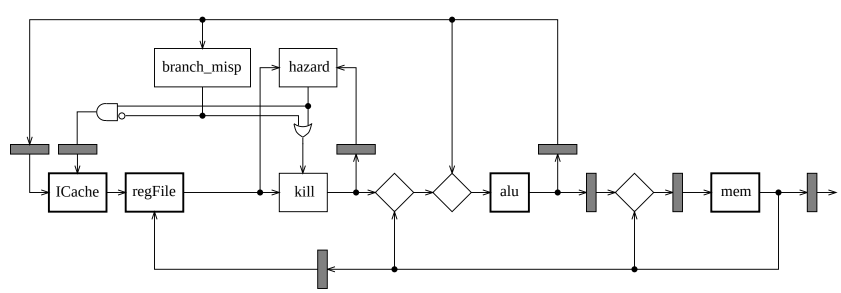


Figure 3.24: Move the `delay` circuit through the third `bypass` circuit using the corresponding time-invariance law

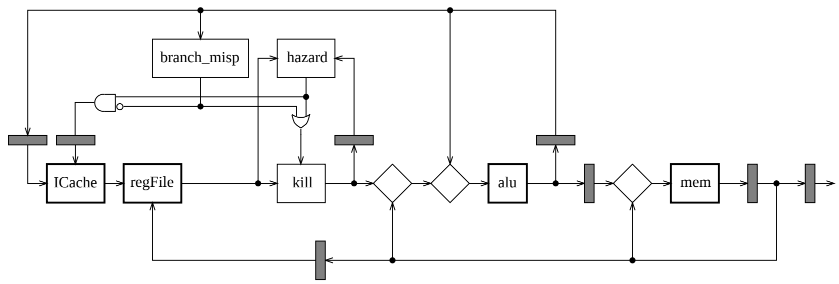


Figure 3.25: Move the `delay` circuit through the `mem` circuit using the corresponding time-invariance law

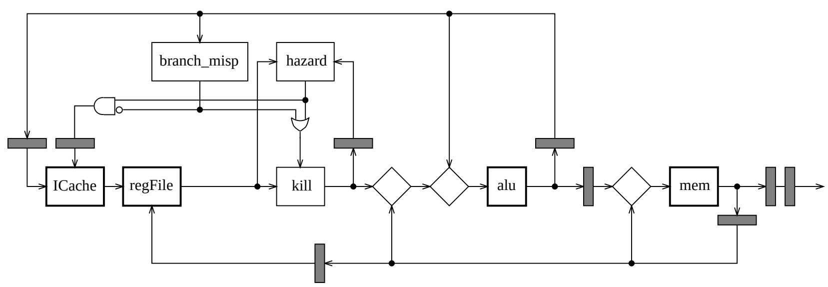


Figure 3.26: Split the `delay` circuit after the `mem` circuit, using the corresponding feedback-rotation law

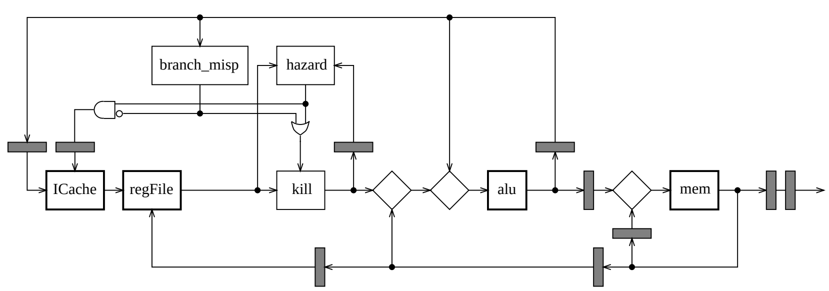


Figure 3.27: Split the `delay` circuit below the `mem` circuit, using the corresponding circuit duplication law

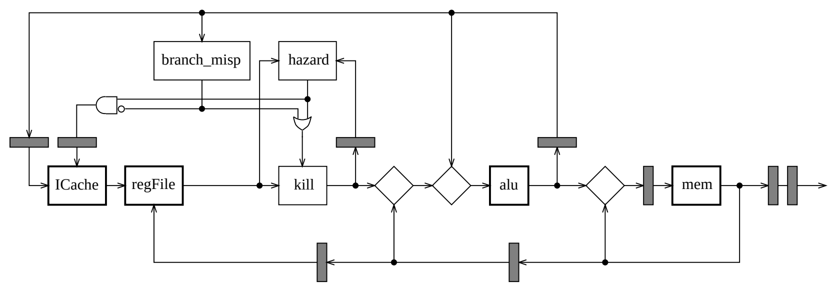


Figure 3.28: Move the `delay` circuit through the last `bypass` circuit, using the corresponding time-invariance law

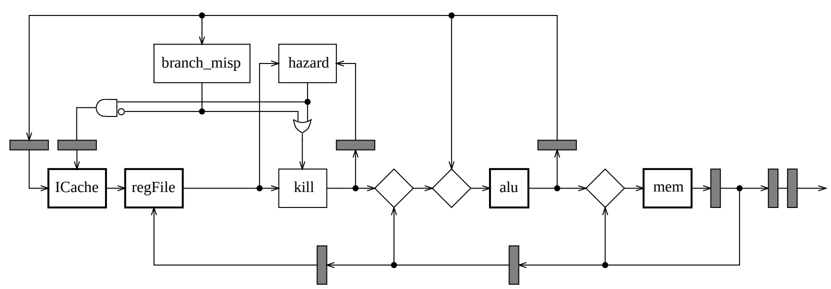


Figure 3.29: Move the `delay` circuit through the `mem` circuit, using the corresponding time-invariance law

We'll keep moving this last delay a bit, to set up for the hazard-bypass law later on.

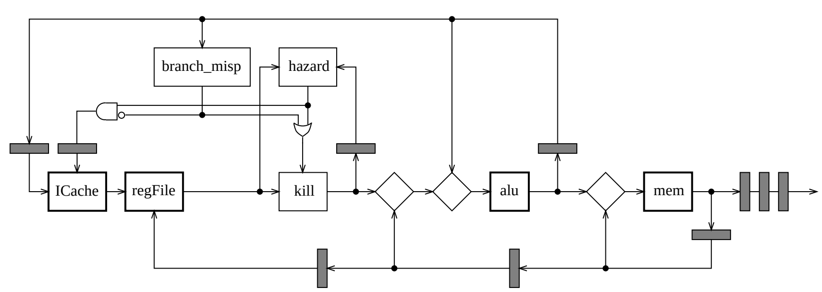


Figure 3.30: Split the `delay` circuit after the `mem` circuit, using the feedback-rotation law

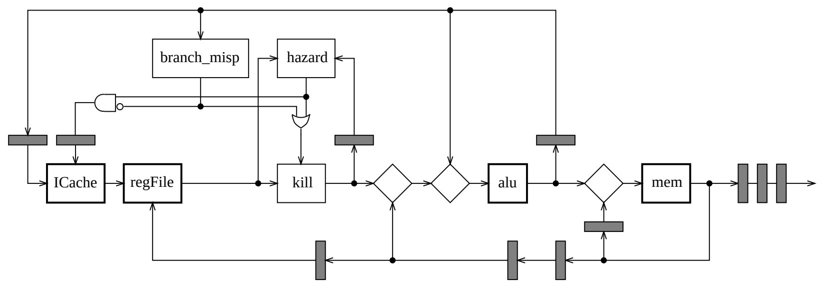


Figure 3.31: Split the bottom-rightmost `delay` circuit, using the circuit duplication law

3.4.2 Move control wires stage

In this stage we move all wires not directly involved with forwarding logic to either before or after all of the `bypass` circuits. This is to enable the hazard-bypass laws, which we apply in a later step. We move the wires by inserting projection circuits and using the corresponding projection-commutativity laws. While we're at it, we'll also insert `proj_ctrl` circuits on the inputs to the `hazard` circuit, so that we can later on move the register file next to the first bypass.

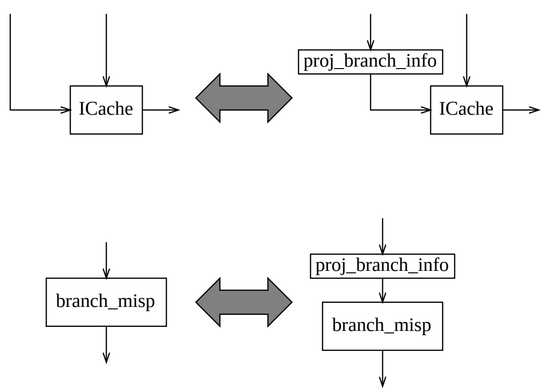


Figure 3.32: Projection insertion laws for `proj_branch_info`

The wire we want to move in this case is the feedback wire after the `alu` circuit, which becomes the input to `branch_misp` and `iCache`. The projection that allows us to move the wire is called `proj_branch_info`. On each clock cycle, `proj_branch_info` examines the opcode field of its input transaction. If it is a branch instruction, then it outputs a transaction with the same opcode, destination register name, destination value, and speculative branch target PC fields as the input transaction, but with all other fields (including source-operand register name fields) set to their default values[2]. If the transaction is not a branch instruction, then `proj_branch_info` outputs `nopTrans`. Since the `iCache` and `branch_misp` circuits only examine branch instructions, and in fact only those fields that `proj_branch_info` lets through to its output, then `proj_branch_info` really is an input projection of these two circuits (Figure 3.32). We thus insert these projections and move them towards the `alu` circuit.

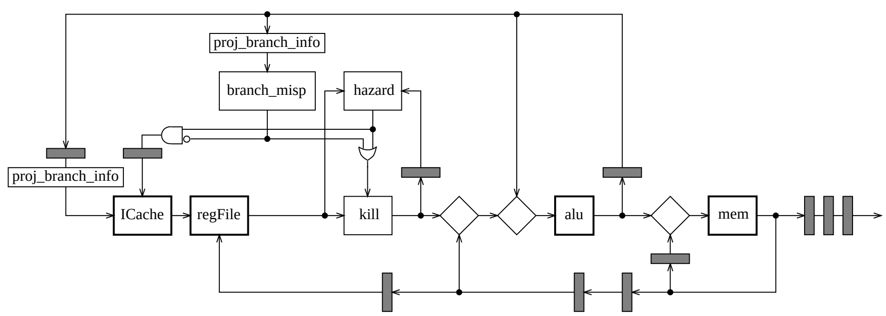


Figure 3.33: Insert `proj_branch_info` projection on the inputs to `iCache` and `branch_misp`, using the corresponding projection laws from Figure 3.32

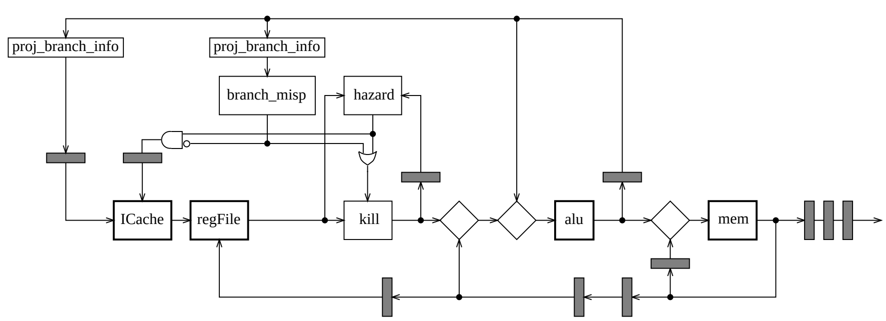


Figure 3.34: Move `proj_branch_info` past the left-most `delay`, using the corresponding time-invariance law

[2] Our ISA architecture hard-wires register `R0` to zero, so `R0` serves as the default value for register names

To continue moving the `proj_branch_info` projection, we apply the circuit duplication law in reverse, merging the two projections into one.

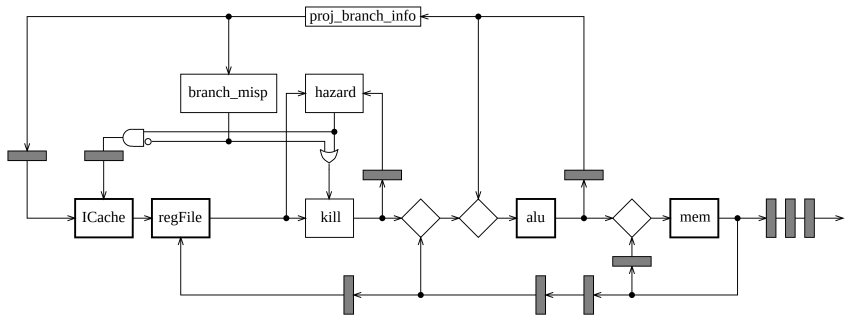


Figure 3.35: Merge the two instances of `proj_branch_info`, using the circuit duplication law in reverse

At this point we can't move the `proj_branch_info` circuit any further, since we cannot insert a `proj_branch_info` circuit on the wire leading to the second bypass without changing the functionality of the pipeline. What we do instead is split the `delay` that is to the right of the projection, using the feedback rotation law (and split the feedback wire while we're at it). Once we have duplicated the `delay`, we can continue moving `proj_branch_info` down towards the `alu` circuit.

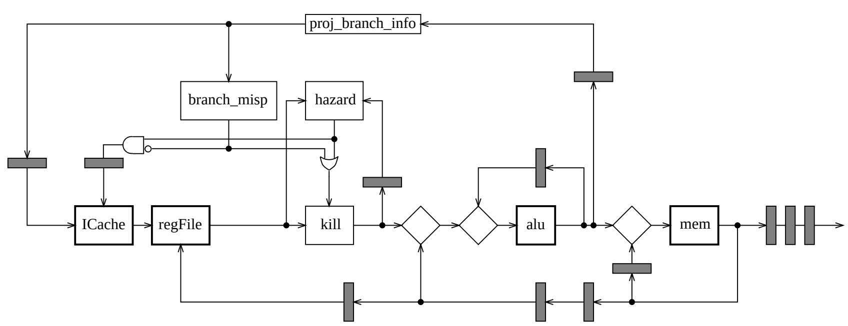


Figure 3.36: Split the `delay` circuit ahead of `proj_branch_info`

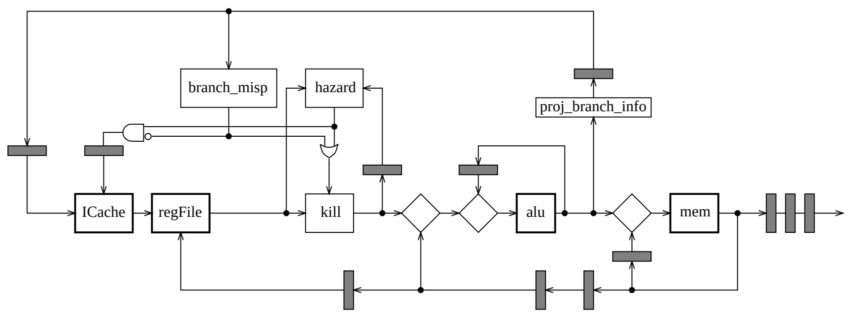


Figure 3.37: Move the `proj_branch_info` circuit past the `delay` circuit using the corresponding time-invariance law

Now that `proj_branch_info` is at the output of the `alu` circuit, we can use *projection-invariance* laws to move the projection to the end of the pipeline. Projection-invariance laws act somewhat like commutativity laws, and state that the output of a projection is unchanged when its input signal is moved across another circuit. Figure 3.38 shows some of the laws for `proj_branch_info`. In particular, we can move the projection past the third `bypass` circuit and the `mem` execution unit of Figure 3.37, since neither of these circuits alter a transaction's branch information.

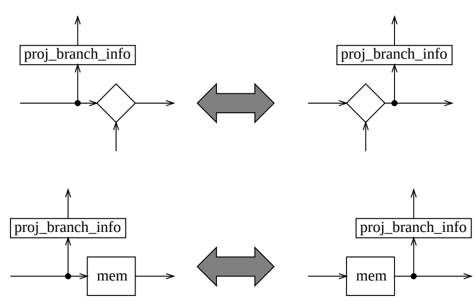


Figure 3.38: Projection-invariance laws for `proj_branch_info`

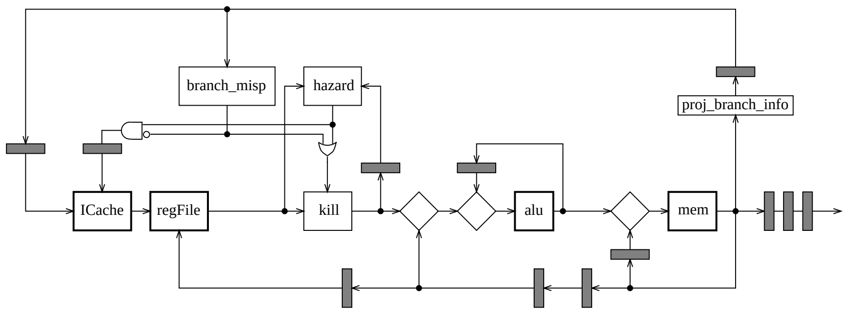


Figure 3.39: Move proj_branch_info past the third bypass and mem circuit, using the projection invariance laws from Figure 3.38

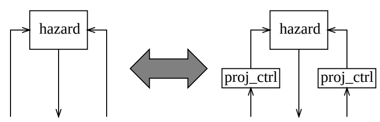


Figure 3.40: proj_ctrl projection insertion law

To prepare for a future stage, we will also add proj_ctrl projections to the inputs of the hazard circuit. The proj_ctrl circuit passes the opcode, source register name, and destination register name fields of its input transaction through unchanged, but zeros-out all other fields. Since the hazard circuit only examines these *control* fields, then the projection insertion law shown in Figure 3.40 is valid.

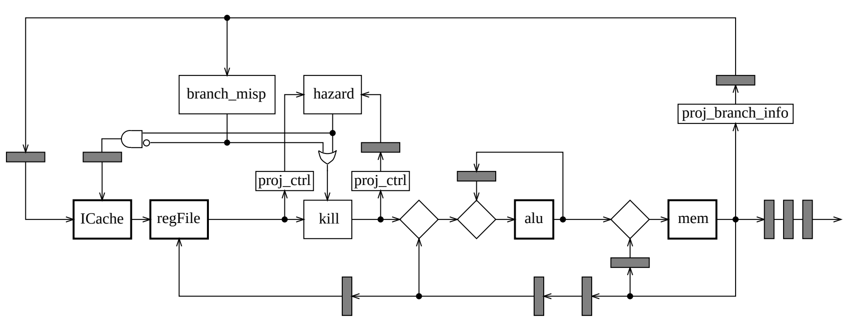


Figure 3.41: Add proj_ctrl projections to the inputs of the hazard circuit using the corresponding projection-insertion laws (Figure 3.40), and move the right-most proj_ctrl circuit past the delay using the corresponding time-invariance law

3.4.3 Propagate hazard information stage

At this point we would like to start removing `bypass` circuits using the hazard-bypass law. But this law can only be applied when there are no hazards between the affected stages. So we generate a no-hazard projection at the end of the dispatch stage (which is justified by a projection-absorption law applicable to the kill-circuit complex in that stage), and then move it between the first and second bypass circuits.

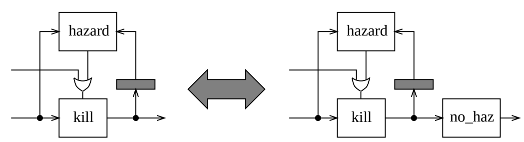


Figure 3.42: Generalized `no_haz` projection insertion law

The `no_haz` projection insertion law shown in Figure 3.42 is a slight generalization of the law discussed in Section 3.3.4. This generalized law holds since the `kill` circuit is still guaranteed to "squash" all potential hazards, and in fact may squash other transactions as well. We use this law to insert a `no_haz` circuit after the `kill` circuit in the microarchitecture.

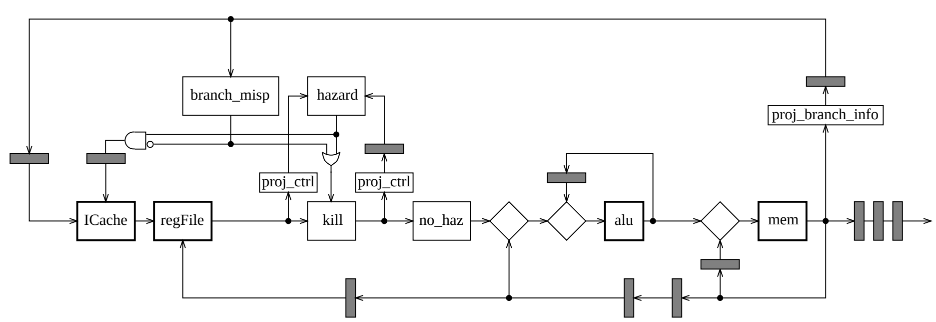


Figure 3.43: Insert a `no_haz` projection after the `kill` circuit, using the projection insertion law shown in Figure 3.42

The `no_haz` projection commutes with `bypass` circuits. One can see this by noting that `bypass` never changes the transaction fields that `no_haz` examines. Thus `no_haz` will squash the same transactions regardless of whether it is placed before or after the `bypass`. If `no_haz` does squash a transaction by replacing it with `nopTrans`, then `bypass` will not modify the squashed transaction, since `nopTrans` contains no source operands. The `no_haz` circuit acts like an identity on transactions it does not squash, so again it does not matter whether it is placed before or after the `bypass` circuit in this case.

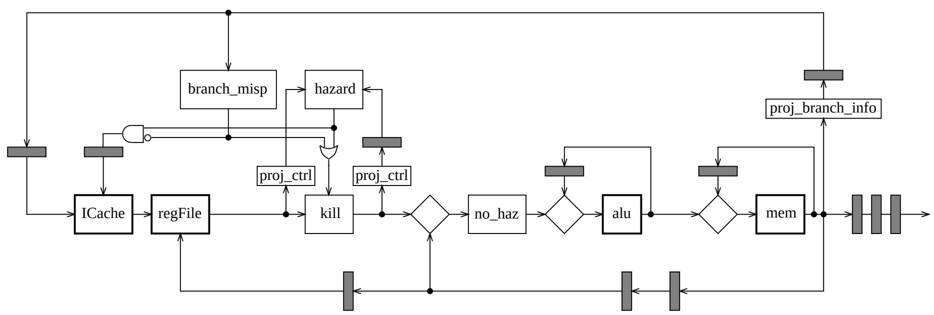


Figure 3.44: Commute `no_haz` with the first `bypass`, using the corresponding projection commutativity law (we also reroute the `mem` stage feedback wire)

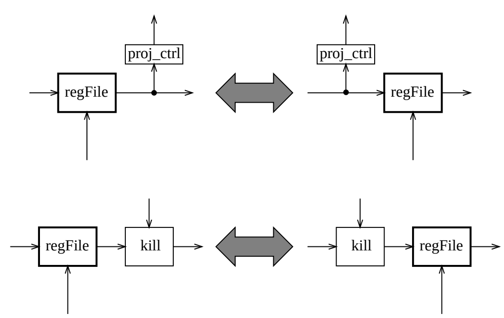


Figure 3.45: register file commutativity laws

We will next swap the register file with the `kill` circuitry using the two laws shown in Figure 3.45, so that the register file is closer to the bypass circuits we want to eliminate. The first law holds since the register file does not modify a transaction's control fields. It is easy to show that the second law holds by performing a case analysis on the Boolean input into `kill`: If the input is `true` at a given clock cycle, then both the left-hand and right-hand circuits output `nopTrans`. If the input is `false`, then the `kill` circuit acts as an identity, so the outputs in both circuits are identical.

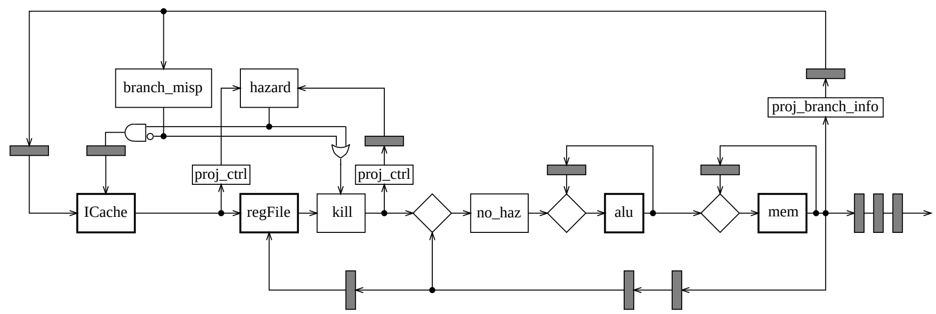


Figure 3.46: Commute the first `proj_ctrl` projection with the register file, using the first law of Figure 3.45

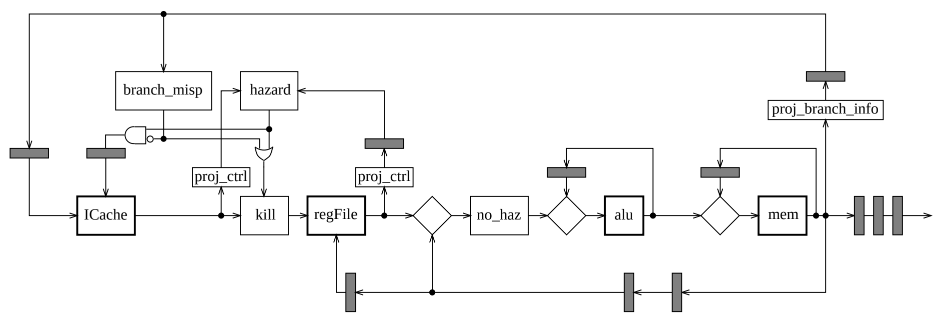


Figure 3.47: Commute the register file with the `kill` circuit, using the second law of Figure 3.45

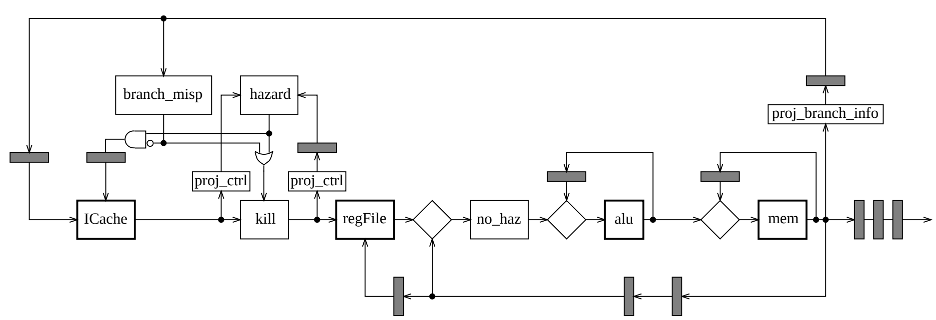


Figure 3.48: Commute the second `proj_ctrl` projection with the register file, using the first law of Figure 3.45

3.4.4 Remove forwarding logic stage

We are now in a position to start removing bypass circuits. The first bypass circuit can be removed immediately, due to the register-bypass law:

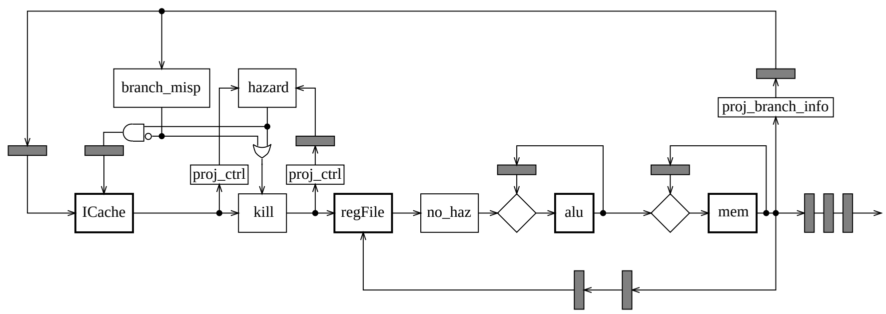


Figure 3.49: Use the register-bypass law to remove the left-most `bypass` and the `delay` circuit below it

We can now apply the hazard-bypass law to remove the bypass circuit just prior to the memory unit.

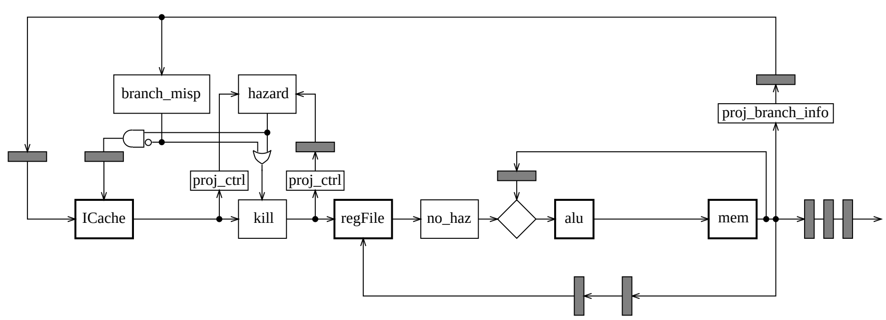


Figure 3.50: Remove the right-most `bypass` circuit using the hazard-bypass law

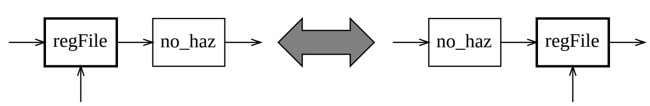


Figure 3.51: register file commutes with hazard projection

Next, we can swap the `no_haz` projection with the register file (Figure 3.51), since the register file never alters its input's control fields, and since the internal state of the register file is only affected by its writeback input, not its data input. Once we have swapped the two components, we can remove the `no_haz` projection by applying the law in Figure 3.42.

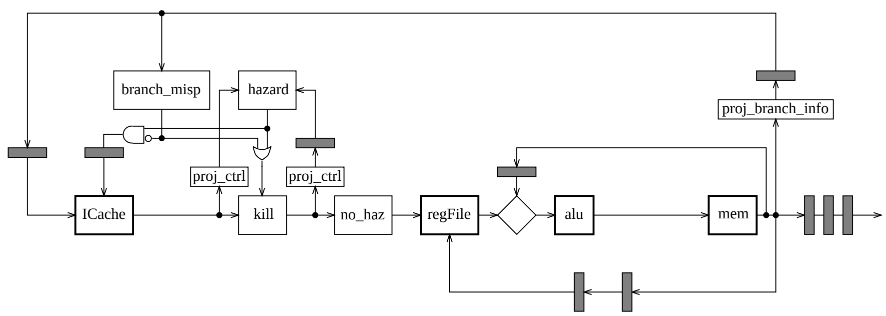


Figure 3.52: Swap the register file with `no_haz`, using the commutativity law in Figure 3.51

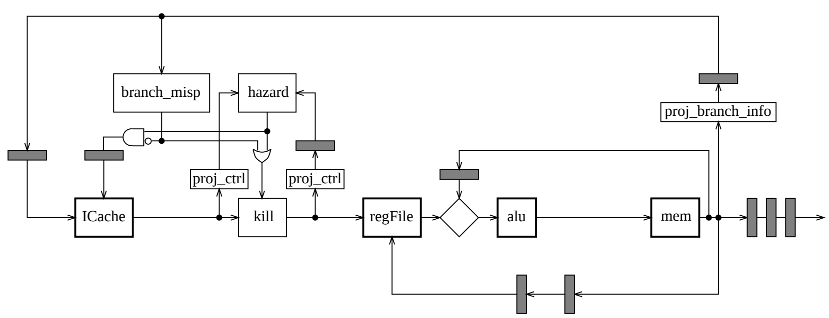


Figure 3.53: Remove `no_haz`, using the `no_haz` projection insertion law (Figure 3.42) in reverse

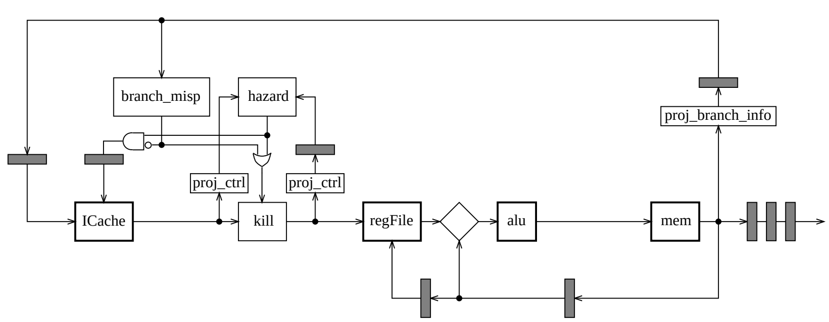


Figure 3.54: Merge the `delay` feeding into the remaining `bypass` circuit with the right-bottom-most delay, using the circuit-duplication law in reverse.

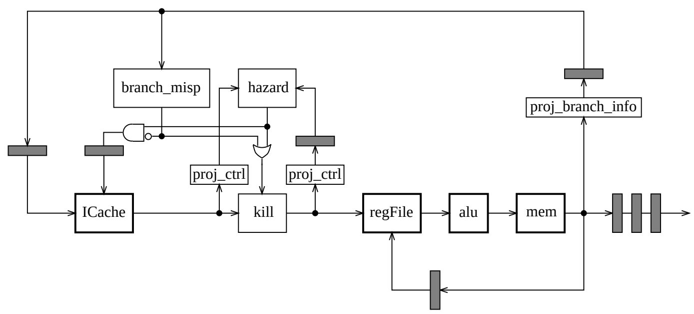


Figure 3.55: Remove the last `bypass` circuit, using the register-bypass law

3.4.5 Cleanup stage

The pipeline has now been simplified as much as possible, except that there are still some extra delay components as well as several unnecessary projection circuits. We merge delay components, then move the projection circuits back to their places of origin and remove them using the projection laws in the opposite direction.

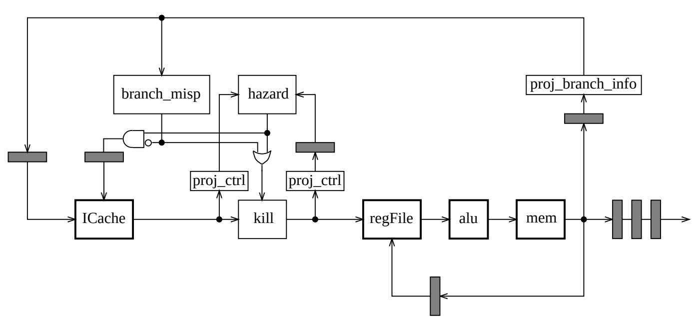


Figure 3.56: Swap the `proj_branch_info` projection with the `delay` next to it, using the corresponding time-invariance law.

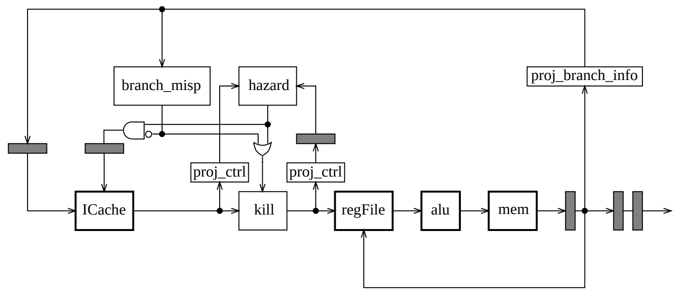


Figure 3.57: Merge the three forking `delay` circuits after the `mem` circuit, using the feedback rotation law in reverse.

We would like to remove as many `delay` circuits as possible when simplifying microarchitectures, and there is a way we can merge the `delay` leading into the `hazard` circuit with the `delay` after the `mem` unit. Neither the `alu` nor the `mem` units ever modify the control fields of a transaction, so `proj_ctrl` commutes with both of them (Figure 3.58).

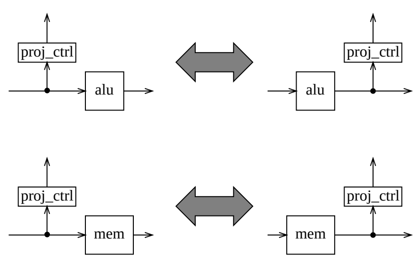


Figure 3.58: More `proj_ctrl` projection invariance laws

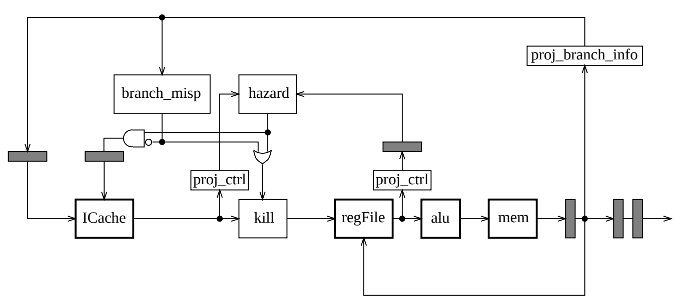


Figure 3.59: Move the right-most `proj_ctrl` circuit past the register file, using the first law of Figure 3.45

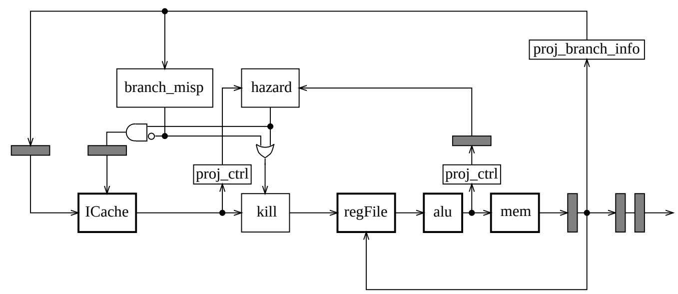


Figure 3.60: Move the right-most `proj_ctrl` circuit past the `alu`, using the first law in Figure 3.58

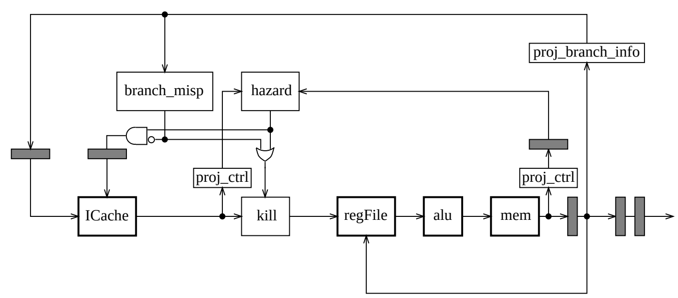


Figure 3.61: Move the right-most `proj_ctrl` circuit past the `mem`, using the second law in Figure 3.58

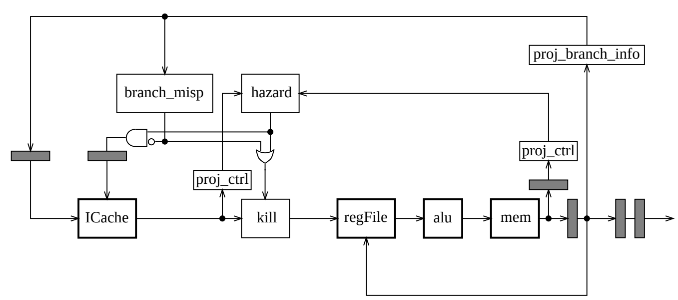


Figure 3.62: Swap the right-most `proj_ctrl` circuit with the `delay`, using the corresponding time-invariance law

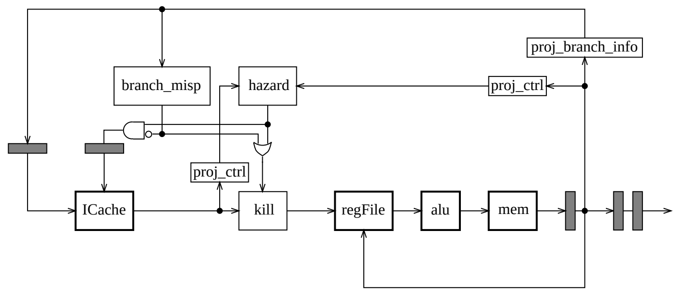


Figure 3.63: Merge the `delay` after the `mem` unit with the `delay` below the right-most `proj_ctrl`, using the feedback rotation law in reverse

All that remains now is to absorb the projection circuits back into the circuits they were created from.

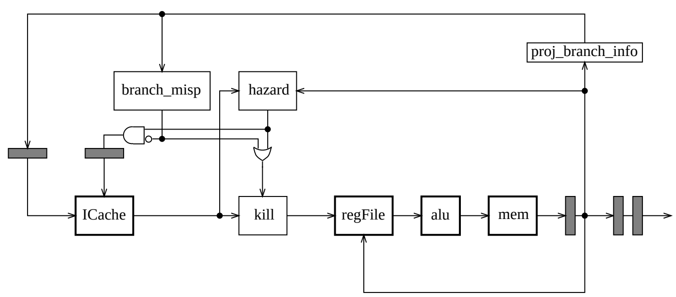


Figure 3.64: Remove `proj_ctrl` circuits, using the projection insertion law of Figure 3.42 in reverse

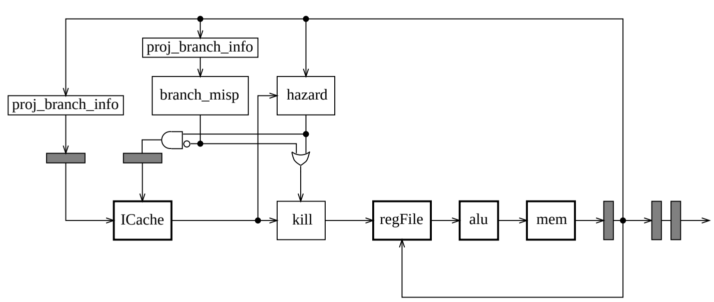


Figure 3.65: Split the `proj_branch_info` projection, using the circuit duplication law

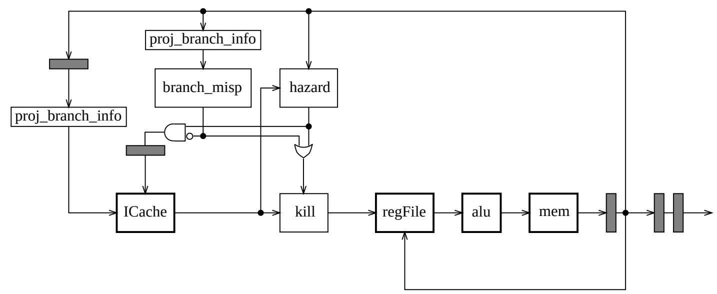


Figure 3.66: Swap the left-most `proj_branch_info` projection with the `delay` circuit below it, using the corresponding time-invariance law

3.4.6 Final pipeline

After removing the `proj_branch_info` projections, we come to the final microarchitecture in Figure 3.67. This circuit still outputs exactly the same transaction values, cycle-for-cycle, as the microarchitecture in Figure 3.10, but is considerably less complex.

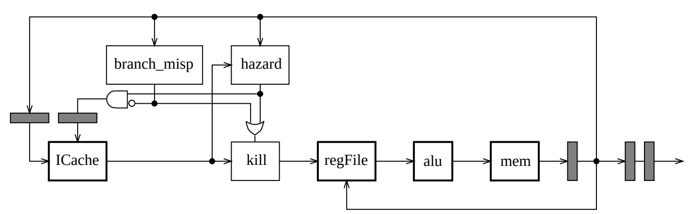


Figure 3.67: The final pipeline, after removing the `proj_branch_info` projections using the projection insertion laws of figure 3.32 in reverse

3.4.7 Verifying the final microarchitecture

We can now apply conventional state machine-based techniques to verify that the final microarchitecture is a valid implementation of its instruction set architecture. Such a verification was recently carried out by Day, Aagaard, and Cook[20] for the microarchitecture of Figure 3.67. Using alternate definitions of the Hawk primitives they were able to automatically translate the Hawk microarchitecture description to an observationally equivalent state machine representation. They then used the *pipeline flushing* methodology of Burch and Dill[11] to complete the verification.

Pipeline flushing method

In the Burch and Dill approach an implementation microarchitecture represented as a state machine is shown to satisfy an instruction set architecture (ISA), also represented as a state machine, by constructing an *abstraction function* that maps the implementation machine's internal state to the internal state of the ISA machine. To verify the implementation machine it must be shown that an abstraction function F maps the initial state of the implementation machine to the corresponding ISA machine's initial state, and that given any reachable implementation machine state s and current input inp, that

$$F\ (Next_{impl}\ s\ inp) = Next_{ISA}\ (F\ s)\ inp \tag{3.1}$$

where $Next_{impl}$ and $Next_{ISA}$ are the next-state transition functions of the implementation and ISA state machines, respectively.

The abstraction function F is constructed by *flushing* the implementation machine. That is, F examines the implementation machine's internal state to determine which instructions have been issued to the pipeline but have not completed yet, and calculates what the final architectural state (i.e. the contents of user-visible registers and memory) would be when those instructions are completed, assuming no new instructions were issued. F can be defined semi-automatically by augmenting $Next_{impl}$ with an extra boolean parameter called *flush*. If *flush* is set then $Next_{impl}$ does not issue a new instruction to the pipeline, but does continue to process in-flight instructions. F is then constructed by iterating the augmented $Next_{impl}$ transition function (with *flush* set to true) until all in-flight instructions have completed. For pipelined architectures, the number of iterations is bounded by the number of pipeline stages.

Day, Aagaard, and Cook constructed an appropriate abstraction function F by augmenting the generated Hawk microarchitecture state machine in this fashion. They then used the automated verification tool SVC[5] to verify that F satisfied equation (3.1). This equation is only required to hold for *reachable* states, that is, implementation machine states obtainable from some series of next-state transitions from the initial state. The authors constructed a predicate P characterizing the set of reachable states, which they gave to SVC as an assumption. They verified that P did in fact characterize the set of reachable states using McMillan's SMV[58] model checker.

Benefit of algebraic simplification

While the authors could have used pipeline flushing to verify the original pipelined microarchitecture of Figure 3.10, they claim that the simplified microarchitecture of Figure 3.67 is less complex, making it more amenable to automated verification.

Chapter 4

Formalizing Hawk in higher order logic

To ensure the correctness of the Hawk transformations we described in Chapter 3, we need to work within a *formal semantics* for Hawk programs. That is, we need to have an unambiguous mathematical interpretation of what a given Hawk program means, as well as a notion of what it means for two Hawk programs to be equivalent.

Since we are mainly concerned with verifying the correctness of microarchitectural laws, rather than fully modeling the Hawk language itself, we have chosen to formalize only a subset of the language. In particular, we have chosen those features of Hawk that can be directly interpreted as elements of higher order logic, as supported by the Isabelle theorem prover. This precludes us from using some of Hawk's more advanced features, such as multi-parameter type classes and nested definitions. Fortunately, the circuits and transformations we consider can be adequately expressed without these features, and in return we gain the full benefit of Isabelle's proof machinery, including its type checker, parser, pretty-printer, and higher-order unification tactics.

Section 4.1 gives a brief and informal account of higher order logic, and assumes the reader is familiar with first order predicate calculus, and the basic concepts associated with typed functional languages, especially the notions of first-class functions and Hindley-Milner type polymorphism. It borrows heavily from material by Melham[64], as well as Gordon and Melham's introduction to another higher order logic theorem prover[29] (also called HOL), the Isabelle reference manual[74], and the chapter on higher order logic in Isabelle's object logics manual[69]. The reader should consult these sources for a more thorough introduction.

4.1 Elements of higher order logic

Higher order logic is a logic of functions. The traditional bifurcation between terms and formulas made in predicate calculus is not present in higher order logic. Instead all operators, including quantifiers and propositional connectives, are represented by (possibly higher order) functions. To avoid logical inconsistencies, a type discipline is imposed on terms, based on a restricted form of Hindley-Milner polymorphism[1].

The use of higher order functions as a first class construct significantly reduces the number of primitive axioms and inference rules in HOL. Many of the primitive syntactic forms in predicate calculus, such as quantifiers and most of the logical operators, are actually derived operators in HOL.

As a result, the kernel of a theorem prover implementing higher order logic can be quite small, as little as a few hundred lines of code in a functional programming language. This has the happy consequence of reducing the likelihood of defects occurring in the overall theorem prover implementation, provided that all proof steps are checked by the kernel.

4.1.1 Terms

Higher order logic terms are built from the following four syntactic entities:

- **Constants.** Examples are *True*, *False*, 0, and *Suc* (the function that takes a number n and returns $n + 1$).

- **Variables.** Elements of this category are drawn from an infinite set of variable names $\mathcal{V}$. Variables can be bound inside function definitions, in contrast to constants, which cannot.

- **Function applications.** Applications are written using juxtaposition (i.e. by separating the function from the argument it is being applied to with spaces). Thus the application of the *Suc* function to the number 3 is written as *Suc* 3.

[1]The main restriction being that only top-level expressions can be given universal types. Thus the term $\mathit{let}\ \mathit{id} = (\lambda\ x.\ x)\ \mathit{in}\ (\mathit{id}\ \mathit{id})$ is not typeable in higher order logic, however the top-level constant definition $\mathit{id} = (\lambda\ x.\ x)$ is typeable with type $'a \Rightarrow\ 'a$, as is the top-level expression $\mathit{id}\ \mathit{id}$.

- **λ-abstractions.** This category corresponds to anonymous functions in a functional programming language. A λ-abstraction denotes a function of one parameter. An example is the function that increments a number by two, written as $(\lambda x.\ \mathit{Suc}\ (\mathit{Suc}\ x))$.

To improve readability, most higher order logic theorem provers allow the user to declare that a given two-argument function constant should be parsed and printed as an infix operator. Thus the term $(+\ 1\ (+\ 3\ 6))$ can be more conveniently read and written as $(1\ +\ 3\ +\ 6)$. To further reduce the number of parentheses needed one can express an operator's associativity and its precedence with respect to other operators. For example, if the user has declared an annotation stating that the multiplication operator has higher precedence than the addition operator, then one can write terms such as $(+\ (*\ 1\ 2)\ (*\ 8\ 3))$ in the more familiar form of $(1\ *\ 2\ +\ 8\ *\ 3)$.

Higher order logic is a "total" language, with a meaning defined for every well-typed term. Constants evaluate to themselves, and the meaning of an application of a λ-abstraction to an argument is given by substitution. Thus the term $(\lambda\, x.\, x\ +\ x)\ (2\ *\ 3)$ is logically equivalent to $(2\ *\ 3)\ +\ (2\ *\ 3)$. Notice that the argument expression $(2\ *\ 3)$ is substituted as is, without first "evaluating" it. One can also substitute expressions containing a mixture of free variables and constants. Substitution in such cases is capture-avoiding, meaning that bound variables in nested λ-abstractions will be renamed if they clash with free variables in the argument being substituted.

4.1.2 Types and type operators

Every HOL term is associated with a type. To begin with, Isabelle HOL assumes an infinite set of type variables $\mathcal{TV}$ (whose elements are typically written $'a$, $'b$, etc), as well as the primitive type constants *bool* and *nat*, corresponding to a two-element set of booleans and the set of natural numbers, respectively.

More complex types can be constructed through the use of *type operators.* The application of a type operator to one or more types is written in postfix form. The only primitive type operator in HOL is *fun*, the function-space operator, which given a domain type τ and range type σ as arguments, denotes the type of functions from τ to σ. The

Isabelle theorem prover provides an infix syntax for the *fun* operator, so that $(\tau, \sigma)\,\mathit{fun}$ can be more conveniently read and written as $\tau \Rightarrow \sigma$. The infix form is right-associative, so that $\tau \Rightarrow \sigma \Rightarrow \rho$ is the same as $\tau \Rightarrow (\sigma \Rightarrow \rho)$. Isabelle also provides several theories containing derived type operators, such as *set* and *list*.

Type polymorphism

It is often the case that a term can be assigned more than one type. For example, the function that returns *True* regardless of its argument, $(\lambda\, x.\, \mathit{True})$, could have type $\mathit{bool} \Rightarrow \mathit{bool}$, but could also have type $\mathit{nat} \Rightarrow \mathit{bool}$ or type $(\mathit{bool} \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$. In fact, for any type τ, the function above could have type $\tau \Rightarrow \mathit{bool}$. Rather than restricting such terms to a single type, one can instead assign them a *polymorphic* type, using type variables. Thus one could associate the type $'a \Rightarrow \mathit{bool}$ to the term, where $'a$ is a variable drawn from $\mathcal{TV}$. By default the Isabelle theorem prover infers the most general such type when constructing terms.

4.1.3 Primitive constants

Pure higher order logic contains only three primitive constants: Implication, equality, and the Hilbert ε-operator (also called *choice*). The constants and their type signatures are shown in Figure 4.1. The meaning of implication and equality correspond to their intuitive meanings in other classical logics: $A \rightarrow B$ is true if and only if either A is false or B is true (or both). The term $x = y$ is true exactly when x is logically equivalent to y.

The third primitive constant is somewhat similar to the axiom of choice in set theory. Given a function P of type $\tau \Rightarrow \mathit{bool}$, then $\mathit{Eps}\ P$ denotes some element x of type τ such that $P\ x$ is true. No other information about x is known. If no such element exists (i.e. P is equal to $(\lambda\, x.\, \mathit{False})$), then $\mathit{Eps}\ P$ denotes a fixed, arbitrary element of type τ. To make choice expressions more readable, they are often written in an alternate syntax using the ε symbol, so that if E is a boolean-valued expression possibly containing occurrences of x, then $\mathit{Eps}\ (\lambda\, x.\, E)$ is written as $\varepsilon\, x.\, E$, and pronounced as "some x such that E holds (if any)".

Table 4.1: The primitive constants of HOL

Constant	Name	Type	Notation
implication	$\rightarrow$	$bool \Rightarrow bool \Rightarrow bool$	$P \rightarrow Q$
equality	$=$	$'a \Rightarrow 'a \Rightarrow bool$	$x\ =\ y$
choice	*Eps*	$('a \Rightarrow bool) \Rightarrow 'a$	$\varepsilon\, x.\, P\ x$

Table 4.2: Some derived constants in Isabelle HOL

Constant	Name	Type	Notation
truth	*True*	$bool$	*True*
falsity	*False*	$bool$	*False*
negation	*Not*	$bool \Rightarrow bool$	$\neg P$
conjunction	*And*	$bool \Rightarrow bool \Rightarrow bool$	$P \wedge Q$
disjunction	*Or*	$bool \Rightarrow bool \Rightarrow bool$	$P \vee Q$
universal quantifier	*All*	$('a \Rightarrow bool) \Rightarrow bool$	$\forall\, x.\, P\ x$
existential quantifier	*Ex*	$('a \Rightarrow bool) \Rightarrow bool$	$\exists\, x.\, P\ x$
unique existence	*Ex*1	$('a \Rightarrow bool) \Rightarrow bool$	$\exists!\, x.\, P\ x$
function composition	*Comp*	$('a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'a) \Rightarrow 'c \Rightarrow 'b$	$f\ \circ\ g$
conditional	*If*	$bool \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$	*if* P *then* x *else* y
let	*Let*	$'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b$	*let* $x{=}e$ *in* $f\ x$

4.1.4 Defined constants

Surprisingly, the above three constants are enough to allow all of the traditional predicate calculus quantifiers and Boolean connectives to be defined as derived constants. The names, types and syntax of the derived constants are given in Figure 4.2.

4.1.5 Inference rules and proofs

Most of the axioms and inference rules of higher order logic correspond to those for predicate calculus. Rather than present them all, in Figure 4.1 we show the additional rules needed to support equality, functions, and choice. Each rule assumes that its constituent terms are well-formed and that all free variables among the predicates are consistently typed. In the rules the letters P and Q stand for boolean-valued terms, R stands for a

$$\frac{P \to Q \qquad Q \to P}{P = Q}\ (= I) \qquad \frac{P = Q \qquad P}{Q}\ (= E)$$

$$\frac{}{a = a}\ (\mathit{refl}) \qquad \frac{a = b}{b = a}\ (\mathit{sym}) \qquad \frac{a = b \qquad b = c}{a = c}\ (\mathit{trans})$$

$$\frac{a = b}{(\lambda x.\ a) = (\lambda x.\ b)}\ (\mathit{abs})^{\dagger} \qquad \frac{f = g \qquad a = b}{f\ a = g\ b}\ (\mathit{comb})$$

$$\frac{}{(\lambda x.\ a) = (\lambda y.\ a[y/x])}\ (\alpha\ \mathit{conv})^{\flat} \qquad \frac{}{((\lambda x.\ a)\ b) = a[b/x]}\ (\beta\ \mathit{conv}) \qquad \frac{f\ x = g\ x}{f = g}\ (\mathit{ext})^{*}$$

$$\frac{R\ a}{R\ (\mathit{Eps}\ R)}\ (\varepsilon\ I)$$

Figure 4.1: Inference rules specific to higher order logic. $^{\dagger}(\mathit{abs})$ holds if x is not free in the assumptions. $^{\flat}(\alpha\,\mathit{conv})$ holds if y is not free in a. $^{*}(\mathit{ext})$ holds if x is not free in the assumptions, f, or g.

predicate term (i.e. a function-valued term returning a boolean), a, b, and c stand for terms of any type, x and y stand for variables of any type, and f and g stand for functions. The intended meaning is that if all of the terms above the bar are provably true, then the predicate below the bar is provably true. If no terms are displayed above the bar, then the conclusion holds unconditionally, and is an axiom.

Proofs

A proof in higher order logic is carried out by "pasting together" existing inference rules and theorems into a tree-like structure. The root of the tree contains the statement being proved, and the leaves contain axioms or pre-proven theorems. The intermediate nodes consist of inference rule instantiations. The root of the proof is drawn at the bottom of the tree, and the leaves at the top. For example, the theorem $g\ ((\lambda\, x.\, f\ x)\ a)\ =\ g\ (f\ a)$ has the following natural deduction proof:

$$\frac{\dfrac{}{g = g}\ (\mathit{refl}) \qquad \dfrac{}{(\lambda\ x.\ f\ x)\ a = f\ a}\ (\beta\ \mathit{conv})}{g\ ((\lambda\ x.\ f\ x)\ a) = g\ (f\ a)}\ (\mathit{comb})$$

Derived rules

One can also build new inference rules in natural deduction style by constructing proofs with undischarged premises. For example, the following derived rule, which we call (β *expand*), is often useful:

$$\frac{(\lambda x.\ P)\ a}{P[a/x]}\ (\beta\ \mathit{expand})$$

The rule states that boolean terms already shown to be true can be β-expanded at the top-level. This rule is valid, since it is the pasting together of rules already known to be valid:

$$\frac{\dfrac{}{(\lambda x.\ P)\ a = P[a/x]}\ (\beta\ \mathit{conv}) \qquad (\lambda x.\ P)\ a}{P[a/x]}\ (=E)$$

Notice that the premise $(\lambda x.\ P)\ a$ of the derived rule occurs as an undischarged premise of the pasting. Any use of (β *conv*) can always be replaced by the corresponding sequence of existing rules.

The converse of this derived rule is also useful

$$\frac{P[a/x]}{(\lambda x.\ P)\ a}\ (\beta\ \mathit{contr})$$

which has a similar derivation. One can use derived inference rules to shorten proofs. For example, we can use (β *expand*) and (β *contr*) to show[2] that $(\varepsilon x.\ x = z) = z$ for free variable z as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{z = z}\ (\mathit{refl})}{(\lambda x.\ x = z)\ z}\ (\beta\ \mathit{contr})}{(\lambda x.\ x = z)\ (\mathit{Eps}\ (\lambda x.\ x = z))}\ (\varepsilon\ I)}{\mathit{Eps}\ (\lambda x.\ x = z) = z}\ (\beta\ \mathit{expand})$$

Without (β *expand*) and (β *contr*) the proof takes three extra steps and is too large to easily fit on this page.

[2]Remember that $(\varepsilon x\ .\ x = z)$ is syntactic sugar for $\mathit{Eps}\ (\lambda x\ .\ x = z)$

4.1.6 Type definitions

While in theory the primitive *bool* and *nat* types and the function space type operator are enough to construct any type of interest, in practice it is often useful to define new types and type operators that are characterized by abstract value constructors and properties only. Higher order logic theorem provers such as Isabelle provide a *type definition* mechanism to define new abstract types and type operators safely, by constructing them as subtypes of existing types.

To define a new type, the user specifies a name T for the new type, a type expression τ composed from existing types, and a membership predicate $P :: \tau \Rightarrow bool$ indicating which elements of τ should represent elements of the new type. The user also has to exhibit a theorem stating that P holds for at least one element of τ, since all types in higher order logic must be non-empty[3]. The type definition package then generates a new type constant with name T, a pair of functions $Rep\_T :: T \Rightarrow \tau$ and $Abs\_T :: \tau \Rightarrow T$, and the following axioms:

$$\forall\,(x :: T).\, P\ (Rep\_T\ x)$$
$$\forall\,(x :: T).\, Abs\_T\ (Rep\_T\ x) \ = \ x$$
$$\forall\,(y :: \tau).\, P\ y \ \rightarrow \ Rep\_T\ (Abs\_T\ y) \ = \ y$$

The axioms state that Abs_T and Rep_T comprise an isomorphism between the elements of the new type and the domain of P. This isomorphism allows the user to prove abstract properties about elements of T in terms of its representation elements of type τ. Once these properties have been proven, the user never need refer to the representation elements. We demonstrate this by example.

The *prod* type operator

As well as types, the user can define new type operators through the same mechanism by parameterizing the type expression τ with type variables. The number of type variables in the type expression τ determines the number of arguments to the type operator.

[3]Non-emptiness is required so that the choice operator (ε) always denotes a meaningful value.

For example, the $('a, 'b)\ prod$ type operator, written as $('a * 'b)$, takes two types $'a$ and $'b$ as arguments, and constructs the type of all ordered pairs $(x :: 'a,\ y :: 'b)$ drawn from the argument types. We can characterize this type abstractly in terms of three functions

$$\begin{aligned} &pair :: 'a \Rightarrow 'b \Rightarrow ('a * 'b) \\ &fst\ \ :: ('a * 'b) \Rightarrow 'a \\ &snd\ :: ('a * 'b) \Rightarrow 'b \end{aligned}$$

and three axioms:

$$\begin{array}{ll} (Fst) & \forall\, x\ y.\ fst\ (pair\ x\ y) \ = \ x \\ (Snd) & \forall\, x\ y.\ snd\ (pair\ x\ y) \ = \ y \\ (ProdEq) & \forall\, (p :: ('a * 'b))\ q.\quad (p \ = \ q) \ = \ (fst\ p \ = \ fst\ q \ \wedge \ snd\ p \ = \ snd\ q) \end{array}$$

Following Melham[63] we can define the $('a * 'b)$ type operator by specifying the operator name as $prod$, the type expression as $'a \Rightarrow 'b \Rightarrow bool$, and the membership predicate P as

$$P \ \equiv \ \lambda\, f.\ \exists\, x\ y.\ \forall\, a\ b.\ f\ a\ b \ = \ (a \ = \ x \ \wedge \ b \ = \ y)$$

$P\, f$ holds for a function $f :: 'a \Rightarrow 'b \Rightarrow bool$ when $f\ x\ y$ is true for exactly one pair of elements $x :: 'a$ and $y :: 'b$. Thus the function f represents the abstract pair (x, y). The theorem

$$P\ (\lambda\, x\ y.\ x \ = \ (\varepsilon\, x.\ False) \ \wedge \ y \ = \ (\varepsilon\, y.\ False))$$

demonstrates that P holds for at least one element of the representation type.

Once the theorem prover has admitted $prod$ as a new type operator, we can define the functions $pair$, fst, and snd as follows:

$$\begin{aligned} &pair \equiv \lambda\, x\ y.\ Abs\_prod\ (\lambda\, a\ b.\ a \ = \ x \ \wedge \ b \ = \ y) \\ &fst\ \ \equiv \lambda\, p.\ \varepsilon\, x.\ \exists\, y.\ (Rep\_prod\ p)\ x\ y \\ &snd\ \equiv \lambda\, p.\ \varepsilon\, y.\ \exists\, x.\ (Rep\_prod\ p)\ x\ y \end{aligned}$$

From these definitions and the generated isomorphism axioms, we can prove the abstract *prod* axioms (*Fst*), (*Snd*), and (*ProdEq*) as theorems. Once proved, it is no longer necessary to explicitly refer to the definitions of *pair*, *fst*, and *snd*.

4.1.7 Datatypes

Using similar tricks to the *prod* type definition above, it is relatively straightforward, though tedious, to create an abstract unit type, as well as type operators for sums, lists, and trees. Structured types such as these are useful enough that several theorem provers have implemented *datatype* definition packages, which allow the user to concisely specify a broad class of inductively structured types and automatically prove their abstract properties as theorems. These packages are patterned after the datatype declaration forms common to typed functional languages such as ML and Haskell.

A datatype declaration consists of a new type name *Ty*, possibly parameterized by type variables $'a_1 \ldots 'a_n$, and a finite list of *constructor* specifications. Each constructor specification consists of a new name C_i and a list of argument types $t_{i,1} \ldots t_{i,k_i}$.

$$
\begin{aligned}
\mathit{datatype}\ ('a_1, \ldots, 'a_n)\ \mathit{Ty}\ =\ & C_1\ t_{1,1}\ \ldots\ t_{1,k_1}\ | \\
& C_2\ t_{2,1}\ \ldots\ t_{2,k_2}\ | \\
& \ldots \\
& C_m\ t_{m,1}\ \ldots\ t_{m,k_m}
\end{aligned}
$$

Each $t_{i,j}$ can either be an existing type, one of the type variables $'a_1 \ldots 'a_n$, or the newly-declared type $('a_1, \ldots, 'a_n)\ \mathit{Ty}$.

Given such a datatype declaration, the Isabelle datatype package automatically generates a new type definition for *Ty* and a new constant definition for each constructor:

$$
\begin{aligned}
C_1\ & :: t_{1,1} \Rightarrow\ \ldots \Rightarrow\ t_{1,k_1} \Rightarrow ('a_1, \ldots, 'a_n)\ \mathit{Ty} \\
C_2\ & :: t_{2,1} \Rightarrow\ \ldots \Rightarrow\ t_{2,k_2} \Rightarrow ('a_1, \ldots, 'a_n)\ \mathit{Ty} \\
\ldots & \\
C_m\ & :: t_{m,1} \Rightarrow\ \ldots \Rightarrow\ t_{m,k_m} \Rightarrow ('a_1, \ldots, 'a_n)\ \mathit{Ty}
\end{aligned}
$$

The package also generates a series of theorems about the constructors, including the fact that no two constructors ever return the same element of Ty, that each constructor of one or more arguments is an injective function, and that together the constructors comprise all of the elements of Ty.

In addition, the package generates a structural induction theorem, allowing the user to prove global properties of the new type. The structural induction theorem states that a predicate $P :: ('a_1, \ldots, 'a_n)\ Ty \Rightarrow bool$ holds for all elements of Ty if for each constructor C_i, the term $P\ (C_i\ x_{i,1}\ \ldots\ x_{i,k_i})$ holds for all $x_{i,1}, \ldots, x_{i,k_i}$. In proving that $P\ (C_i\ x_{i,1}\ \ldots\ x_{i,k_i})$ holds, it is assumed that $P\ x_{i,j}$ already holds for each argument $x_{i,j}$ of type $('a_1, \ldots, 'a_n)\ Ty$.

List datatype

As an example, the type of finite $'a$ lists can be defined by the following datatype declaration

$$\begin{aligned} \mathit{datatype}\ 'a\ \mathit{list}\ &= \mathit{Nil}\ |\\ &\quad \mathit{Cons}\ 'a\ ('a\ \mathit{list}) \end{aligned}$$

with Nil representing the empty list and $Cons\ x\ xs$ representing the list constructed from head element $x :: 'a$ and tail list $xs :: 'a\ list$. Thus the list $[1, 2, 3]$ of the first three positive natural numbers is represented by the expression $Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))$ of type $nat\ list$.

From the *list* datatype declaration, the datatype package generates the following information:

- A new type operator definition with name $'a\ list$,
- Constant definitions for the constructors

$$\begin{aligned} &\mathit{Nil} && :: 'a\ \mathit{list}\\ &\mathit{Cons} && :: 'a \Rightarrow 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list} \end{aligned}$$

- A theorem stating that Nil and $Cons$ always return separate $'a\ list$ elements

$$\forall\, x\; xs.\; Nil \;\neq\; Cons\; x\; xs$$

- A theorem stating that *Cons* is an injective function

$$\forall\, x\; y\; xs\; ys.\; (Cons\; x\; xs \;=\; Cons\; y\; ys) \;=\; (x \;=\; y \;\wedge\; xs \;=\; ys)$$

- A theorem stating that together *Nil* and *Cons* generate all the elements of type *'a list*

$$\forall\, (xs \;::\; {'a\; list}).\; xs \;=\; Nil \;\vee\; (\exists\, y\; ys.\; xs \;=\; Cons\; y\; ys)$$

- A structural induction theorem for proving global properties of *'a list* elements.

$$\begin{array}{l}\forall\, (P \;::\; {'a\; list} \Rightarrow bool)\; (xs \;::\; {'a\; list}).\\ \quad P\; Nil \;\wedge\; (\forall\, y\; ys.\; P\; ys \;\rightarrow\; P\; (Cons\; y\; ys)) \;\rightarrow\; P\; xs\end{array}$$

The *'a list* type could alternatively be defined directly in terms of existing types, using the type definition package. However, it would be quite a bit of work to manually verify the necessary *list* properties.

Soundness of datatype definitions

When generating a new datatype definition, a theorem prover could simply create the needed datatype properties as axioms. However in practice most datatype packages construct new datatypes conservatively by invoking the theorem prover's underlying type definition facility. A representation predicate for the datatype and a set of function definitions corresponding to the datatype constructors is fashioned such that the desired datatype properties can be proven by the package as theorems. In this way the consistency of the logic is guaranteed to be preserved.

4.2 The Isabelle theorem prover

Many of the proofs in this thesis have been checked by the *generic* theorem prover Isabelle (which we have already referred to in passing). Rather than supporting a single logic, a generic theorem prover is designed to support several logics by instantiating custom provers from a reusable set of program modules. This is based on the observation that many components of a theorem prover, such as parsing, pretty printing, theory management, rewriting tactics, etc. do not particularly depend on the actual logic used. Building a theorem prover able to tackle large verification tasks requires a substantial amount of infrastructure, so it is beneficial to reuse common tools when possible. Isabelle has been instantiated for several logics, including Zermelo-Fraenkel set theory[22], higher order logic, and domain theory.

4.2.1 Certifying proofs in Isabelle

Isabelle is derived from the Cambridge LCF system and follows the LCF approach to certifying proofs[4]. In this approach the user interface to the theorem prover is an interactive read-eval-print loop to the programming language ML. Axioms and theorems are represented as elements of an abstract data type called `thm`. The inference rules of higher order logic are represented as ML functions that return elements of type `thm`. The premises of an inference rule become parameters of the associated ML function.

The user creates new theorems by calling ML procedures, either interactively or from a batch file. The static type system of ML ensures that only the axioms and `thm`-returning functions of the `thm` abstract data type can be used to build new theorems. However, the user can automate common patterns of inference by defining ML procedures, called *tactics*, that use existing `thm` functions and values. These tactics are themselves first class `thm`-returning functions that can be used to build even more powerful tactics, and so on. In this way very high level tactics that perform thousands of primitive inferences can be invoked to certify large proofs securely. To illustrate this approach, we provide a few of Isabelle's axioms and inference rules for higher order logic. In these examples we represent Isabelle terms as strings for readability. In practice terms are built from an algebraic ML datatype `cterm`.

[4]The LCF systems have had a profound influence on the design of both higher order logic theorem provers and modern typed functional languages. Gordon[27] provides an historical account of LCF and the theorem provers influenced by it.

```
"True" : thm

"(x::'a) = x" : thm

"(P::bool --> Q) --> (Q -->P) --> (P = Q)" : thm

beta_conversion : cterm -> thm

transitive : thm -> thm -> thm
```

The first three expressions are axioms. The fourth expression is a function corresponding to the (β *conv*) inference rule. Given a `cterm` of the form `"(λx. a) b"`, the `beta_conversion` function returns a `thm` of the form `"(λx. a) b = a[b/x]"`. The function dynamically checks that its `cterm` argument is a lambda abstraction applied to an argument, and that the `cterm` is well-typed according to the type rules of higher order logic. If these conditions do not hold then `beta_conversion` raises an exception instead of returning. The `transitive` function corresponds to the (*trans*) inference rule. It takes two equational `thm` arguments of the form `"a = b"` and `"b = c"`, respectively. The function checks that both arguments are in fact equations, and that they have the common term `b`. If the checks succeed then `transitive` returns the theorem `"a = c"`.

4.2.2 Higher level tactics

Proofs are constructed by connecting inference rules, axioms and theorems together in some focused way. Patterns of proof construction are called *tactics*.

Isabelle provides a wealth of tactics, ranging from the primitive inference rules exported by the `thm` abstract data type to tactics that rewrite a theorem according to a list of already-proven equations[5], perform prolog-style proof search, and allow the user to interactively prove theorems in a goal-directed fashion. Such high level tactics are essential to carry out verifications of any reasonable size. For example the function

```
simplify : simpset -> thm -> thm
```

[5]Isabelle also provides a primitive rewriting tactic as part of the `thm` abstract data type for efficiency

is one of Isabelle's rewriting tactics. It takes a `simpset`, which is a collection of equation theorems indexed by the structure of their left hand sides for rapid pattern matching, and a `thm` to rewrite against. It repeatedly rewrites the theorem using the the equations stored in the `simpset` as left-to-right rewrite rules until no more equations match any of the theorem's subterms. The `simplify` function then returns the reduced theorem.

Readability of Isabelle proofs

One disadvantage of the LCF approach to certifying theorems is that the structure of the proof itself is not evident, as it is in an English description. Even proofs carried out using primitive tactics contain very little readable proof structure. For example, the primitive proof of the theorem $(\varepsilon\, x.\, x = t) = t$ is given as the following ML expression in Isabelle:

```
refl RS (read_instantiate [("P","(%x. x = ?t)")] selectI)
```

For this reason we will present subsequent higher order logic proofs in English, rather than as Isabelle expressions.

4.3 Embedding Hawk

Given a formal mathematical basis such as higher order logic, there are two common methods for formalizing a programming language such as Hawk within the logic, termed *shallow embedding* and *deep embedding*[10].

Shallow embeddings

In a shallow embedding programming language elements are modeled directly as corresponding elements within higher order logic. Thus programming language types are modeled as types within the logic, programming language numbers as logical numbers, programming language functions as logical functions, and so on.

A shallow embedding works well when the language features being modeled are already present within the logic. In this case all of the logical rules for type checking and proving equality of expressions can be used as is. A disadvantage is that there are typically many more logical functions than there are programming language functions. For example, it is relatively easy in higher order logic to specify the function that solves the halting problem. These "extra" functions usually make it impossible to prove global properties about the programming language being modeled. Another disadvantage occurs with respect to the type system of a language. One often wants to prove global properties of the form "for all types τ, every program of type τ has property $X\ldots$". In many cases proofs of such properties require the use of case analysis or induction over all types, but typically this cannot be done within the logic (though see Völker[91]).

Deep embeddings

A deep embedding consists of one or more inductively defined datatypes representing the abstract syntax of the programming language, and a *meaning function* (or more generally a relation) that maps syntactic elements to logical (semantic) elements. In effect, one builds an interpreter for the language being embedded. One way to determine whether an embedding of a language is shallow or deep is to ask how programming language variables are modeled. In a shallow embedding, language variables become variables of the logic; in a deep embedding, language variables become constants of the datatypes representing the abstract syntax.

A deep embedding allows one to prove global properties by induction over the datatypes representing the abstract syntax of the language. For example, a deep embedding can often be used to prove that all programs in the language are computable, or that all well-typed programs never generate runtime type errors.

Another advantage of a deep embedding is its ability to model language features not present in the logic. For instance, the Haskell programming language has a sophisticated notion of overloading based on *type classes*. While the higher order logic employed by Isabelle implements single parameter type classes, it does not have support for Haskell's multi-parameter or constructor classes. These advanced type class features can only be modeled in Isabelle through a deep embedding.

The primary disadvantage of deep embeddings is the low level at which the language is specified, and the lack of built in theorem proving support for even the simplest operations. All type checking, parsing, pretty-printing, α-conversion and β-conversion of functions, and evaluation of expressions has to be programmed into the theorem prover as part of the embedding. A well-developed theorem prover like Isabelle has a great deal of specialized code for performing inference over its native logic, such as specialized unification and rewriting tactics, heuristically guided proof search routines, and so on. These routines either cannot be used on deeply embedded expressions, or have to be manually refitted. Also, since embedded language expressions are encoded as abstract syntax datatypes *within* the logic, there is an extra level of interpretive overhead when calling inference routines on them.

Embedding Hawk

In this thesis we are primarily interested in proving equivalence between specific microarchitecture components, rather than demonstrating global properties over all possible Hawk programs. In addition, almost the entire subset of Haskell's features that are needed to implement these components are already present in higher order logic. For these reasons we have pursued a shallow embedding of Hawk.

4.4 Modeling recursive definitions

The one critical feature of Hawk that higher order logic does not directly support is the ability to define recursive values, such as signals.

In general, a recursive definition is given by one or more equations, with the function (or value) being defined on the left hand side of each equation and an expression, possibly containing an instance of the function being defined, on the right hand side.

Unlike most programming languages, Isabelle does not normally allow users to create arbitrary recursive definitions, since doing so could easily lead to false theorems. For instance, suppose that Isabelle allowed the following recursive function definition:

$$f :: nat \rightarrow nat$$
$$f\, x = f\, x + 1$$

Isabelle would then add the above equation as a new theorem. But we could then subtract $f\,x$ from both sides to conclude that $0 = 1$ is also a theorem, which is clearly inconsistent.

4.4.1 Axiomatic definitions

Isabelle does allow the user to assume the truth of an arbitrary Boolean formula by declaring it as a new axiom of a theory. Using this facility, the user could create a new theory and specify a recursive Hawk definition as a series of equational axioms. It would then be the user's responsibility to show outside of the logic that all of the axioms are consistent. However, since we want to ensure a high level of confidence in the correctness of our microarchitecture laws, we would prefer a mechanism that could be verified completely within the logic, and thus be checked by Isabelle itself.

4.4.2 Well-founded recursion

Rather than specify recursive functions by possibly inconsistent axioms, Isabelle and several other higher order logic (HOL) theorem provers[29, 73, 81] provide *well-founded* recursive function definition packages, where new functions can be defined conservatively. Recursive functions are defined by giving a series of pattern matching reduction rules, and a well-founded relation.

For example, the *map* function applies a function f pointwise to each element of a finite list. This function can be recursively defined in Isabelle by the following equations:

$$
\begin{array}{ll}
\multicolumn{2}{l}{map :: (\alpha \to \beta) \to \alpha\, list \to \beta\, list} \\
map\, f\, [] & = [] \\
map\, f\, (x\#xs) & = (f\, x)\, \#\, (map\, f\, xs)
\end{array}
$$

The first rule states that *map* applied to the empty list, denoted by [], is equal to the empty list. The second rule states that *map* applied to a list constructed out of the head element x and tail list xs, denoted by $x\#xs$, is equal to the list formed by applying f to x and $map\, f$ to xs recursively.

To define a function using well-founded recursion, the user must also supply a *well-founded relation* on one of the function's arguments[6]. A well-founded relation ($<$) is a relation with the property that there exists no strictly decreasing infinite sequence of elements $x_1, x_2, x_3, x_4, \ldots$.

Given a well-founded relation the recursive definition package checks each reduction rule, ensuring every recursive call on the right-hand side of the rule is applied to a smaller argument than on the left-hand side, according to the relation.

In the case of *map*, we can supply the well-founded relation

$$xs < ys \equiv \mathit{length}\, xs < \mathit{length}\, ys$$

The relation holds when the number of elements in the relation's left-hand list argument is less than the number of elements in the relation's right-hand argument. The definition of *map* contains only one recursive rule, and it is easy to prove that the xs argument of the recursive call of *map* is smaller than the $(x\#xs)$ argument on the left-hand side of the rule, according to this relation. In general, well-founded relations ensure that there are no infinite chains of nested recursive calls.

4.4.3 Coinductive types and corecursive functions

Although well-founded recursion is a useful definition technique, there are many recursive definitions that fall outside its scope (including most of the recursively defined circuits in Hawk). For instance, there is a non-inductive type of *lazy lists* in the Isabelle[73] theorem prover, denoted by $\alpha\, \mathit{llist}$, that is the set of all finite and infinite lists of type α. The function *lmap* over this type is uniquely specified by the following recursive equations[7]:

$$\begin{aligned} \mathit{lmap}\, f\, [] &= [] \\ \mathit{lmap}\, f\, (x\#xs) &= (f\, x)\, \#\, (\mathit{lmap}\, f\, xs) \end{aligned}$$

lmap cannot be defined using well-founded recursion since the length of an infinite list does not decrease upon taking its tail. In fact, the expression

[6] Some well-founded recursion packages only allow single-argument functions to be defined. In this case one can gain the effect of multi-argument curried functions by tupling.

[7] Isabelle uses a different syntax for lazy lists than for finite lists. In this dissertation we use the same syntax for both types.

$lmap\ f\ (x_1 \# x_2 \# x_3 \# \ldots)$ can be unfolded using the above rules to an infinite chain of recursive calls:

$$
\begin{aligned}
& lmap\ f\ (x_1 \# x_2 \# x_3 \# ...) \\
= & \\
& (f\ x_1) \# (lmap\ f\ (x_2 \# x_3 \# ...)) \\
= & \\
& (f\ x_1) \# (f\ x_2) \# (lmap\ f\ (x_3 \# ...)) \\
= & \\
& (f\ x_1) \# (f\ x_2) \# (f\ x_3) \# (lmap\ f\ (...)) \\
= & \\
& ...
\end{aligned}
$$

Defining functions corecursively

The $\alpha\, llist$ type is an example of a coinductive type. Although there is no general induction principle for coinductive types, one can use principles of coinduction to show that two coinductive values are equal, and one can build coinductive values using *corecursion*.

In Isabelle's theory of lazy lists[75], for instance, potentially infinite lists are built through the *llist_corec* operator, which has type $\beta \rightarrow (\beta \rightarrow unit + (\alpha * \beta)) \rightarrow (\alpha\, llist)$. The *llist_corec* operator uniquely satisfies the following recursion equation:

$$
llist\_corec\, b\, g = \begin{cases} [], & \text{if } g\, b = \mathrm{Inl}\,() \\ (x \# (llist\_corec\, b'\, g)), & \text{if } g\, b = \mathrm{Inr}\,(x, b') \end{cases}
$$

The *llist_corec* operator takes as arguments an initial value b and a function g. When g is applied to b, it either returns $\mathrm{Inl}\,()$, indicating that the result list should be empty, or the value $\mathrm{Inr}\,(x, b')$, where x represents the first element of the result list, and b' represents the new initial value to build the rest of the list from. Function g is called iteratively in this fashion, constructing a potentially infinite list.

Using *llist_corec*, we can define *lmap* corecursively as follows:

$$lmap\ \ f\ xs \equiv llist\_corec\ xs\ (map\_head\ f)$$

where

$$\begin{aligned}
&\mathit{map_head} \;\; :: (\alpha \to \beta) \to \alpha\, \mathit{llist} \;\to (\mathit{unit} + (\beta * \alpha\, \mathit{llist}))\\
&\mathit{map_head}\; f\; \mathit{xs} \;\equiv \text{case}\; \mathit{xs}\; \text{of}\\
&\qquad\qquad\qquad\qquad\quad [] \qquad\quad\;\; \Rightarrow \text{Inl}\,()\\
&\qquad\qquad\qquad\qquad \mid\; (x\#\mathit{xs}') \;\Rightarrow \text{Inr}\,(f\,x, \mathit{xs}')
\end{aligned}$$

We could then prove by coinduction that this definition satisfies *lmap*'s recursive equations. Needless to say, this is not the most intuitive specification of *lmap*, and most people would prefer to specify such functions using recursion, if possible. More importantly, corecursive definitions do not match the recursive style of Hawk specification we have developed so far.

4.5 Defining recursive functions as fixed points

In the remainder of this chapter and continuing in Chapter 5 we will present a more general approach that will allow us to define functions such as *lmap* recursively. The basic steps required in our framework to prove that a set of recursive equations is well defined in higher order logic are as follows. The use must:

- Express the recursive equations as a *fixed point* of a functional F.
- Show that for any two different potential solutions supplied to F, F maps them to two potential solutions that are closer together, in a suitable sense.
- Invoke the main result (Section 5.3) to show that the above property of F is sufficient to guarantee that there is a unique solution to the original set of recursive equations.

In this section we deal with the first step.

4.5.1 Unique fixed points

We can convert a system of pattern matching recursive equations into a functional form by employing a standard technique from domain theory[32, 90]. We start by recasting the equations as a single recursive equation using argument destructors or nested case-expressions. For example, the recursive equations defining the *lmap* function are equivalent to the following single recursive equation:

$$
\begin{aligned}
\mathit{lmap}\ f\ l = \text{case}\ l\ \text{of}\quad & \\
& [] \qquad\qquad \Rightarrow [] \\
& \mid (x \# xs) \quad \Rightarrow (f\ x) \# (\mathit{lmap}\ f\ xs)
\end{aligned}
$$

Given f, we can reify this pattern of recursion into a non-recursive functional F of type $(\alpha\, \mathit{llist} \rightarrow \beta\, \mathit{llist}) \rightarrow (\alpha\, \mathit{llist} \rightarrow \beta\, \mathit{llist})$ that takes a function parameter $\mathit{lmap\_f}$:

$$
\begin{aligned}
F\ \mathit{lmap\_f} = \lambda l\,.\ \text{case}\ l\ \text{of}\quad & \\
& [] \qquad\qquad \Rightarrow [] \\
& \mid (x \# xs) \quad \Rightarrow (f\ x) \# (\mathit{lmap\_f}\ xs).
\end{aligned}
$$

Using the recursive equations for lmap, it is easy to show that $\mathit{lmap}\ f = F\,(\mathit{lmap}\ f)$. The value $\mathit{lmap}\ f$ is called a *fixed point* of F. In general, an element x of type α is a fixed point of a function g of type $\alpha \rightarrow \alpha$ if $x = g\,x$. A function may have many fixed points, or none at all. Considering g as a functional representation of a system of recursive equations, each fixed point of g represents a valid solution to the system. If the function g has exactly one fixed point x, then we can think of g as *defining* the value x. We use Hilbert's choice operator (ε) to formalize this notion in HOL:

$$
\begin{aligned}
& \text{fix} :: (\alpha \rightarrow \alpha) \rightarrow \alpha \\
& \text{fix}\, g \equiv \varepsilon x\,.\ x = g\,x \wedge (\forall\, y\, z\,.\, y = g\,y \wedge z = g\,z \longrightarrow y = z)
\end{aligned}
$$

The expression $\text{fix}\, g$ represents the unique fixed point of g, when one exists. If g does not have a *unique* fixed point, then $\text{fix}\, g$ denotes an arbitrary value.

4.5.2 Properties of unique fixed points

As an aside, several nice properties hold when one can establish that a system of recursive equations has a unique solution. For example, unique fixed points can sometimes "absorb" functions applied to other fixed points.

Lemma 1 *Given functions $F : \alpha \rightarrow \alpha$, $G : \beta \rightarrow \beta$, $f : \alpha \rightarrow \beta$, and value $x : \alpha$, such that x is a (not necessarily unique) fixed point of F, G has unique fixed point $\text{fix}\, G$, and $f \circ F = G \circ f$, then $f\, x = \text{fix}\, G$.*

Proof: We have $f\,x = f\,(F\,x) = G\,(f\,x)$. Thus $f\,x$ is a fixed point of G. Since G's fixed point is unique, then $f\,x = \mathrm{fix}\,G$ $\square$

Unique fixed points can also be "rotated", in the following sense:

Lemma 2 *If the composition of two functions $g : \beta \to \alpha$ and $h : \alpha \to \beta$ has a unique fixed point* $\mathrm{fix}\,(g \circ h)$*, then $h \circ g$ also has a unique fixed point, and* $\mathrm{fix}\,(g \circ h) = g\,(\mathrm{fix}\,(h \circ g))$.

Proof: We first note that $h\,(\mathrm{fix}\,(g \circ h)) = h\,((g \circ h)\,(\mathrm{fix}\,(g \circ h))) = (h \circ g)\,(h\,(\mathrm{fix}\,(g \circ h)))$. Thus $h\,(\mathrm{fix}\,(g \circ h))$ is a fixed point of $h \circ g$. Next, suppose that x is an arbitrarily chosen fixed point of $h \circ g$. Then $g\,x = g\,((h \circ g)\,x) = (g \circ h)\,(g\,x)$. Thus $g\,x$ is a fixed point of $g \circ h$. Since $g \circ h$ has a unique fixed point, then $g\,x = \mathrm{fix}\,(g \circ h)$. Applying h to both sides of this equation, we have $h\,(g\,x) = h\,(\mathrm{fix}\,(g \circ h))$. Since x is a fixed point of $h \circ g$, we can reduce the above equation to $x = h\,(\mathrm{fix}\,(g \circ h))$, which demonstrates that the fixed point of $h \circ g$ is unique. Using the definition of fix, we have $\mathrm{fix}\,(h \circ g) = h\,(\mathrm{fix}\,(g \circ h))$. Applying g to both sides of this equation and using the unique fixed point property of $g \circ h$, we conclude that $g(\mathrm{fix}\,(h \circ g)) = \mathrm{fix}\,(g \circ h)$ $\square$

Although we will not use Lemma 1 or Lemma 2 explicitly, they justify many of the graphical transformations that have been undertaken in Chapter 3. In the next chapter we will show how to find unique fixed point solutions to recursive function definitions in a manner that can be semi-automated in Isabelle.

Chapter 5

Converging equivalence relations

While unique fixed points are a useful definition mechanism, it can be difficult to show that they exist for a given function. A direct proof usually involves constructing an explicit fixed point witness using other definition techniques, such as corecursion or well-founded recursion. Little effort seems to be saved.

We propose an alternative proof technique, based on concepts from domain theory[32, 90] and topology[12, 80] where one builds a collection of ever-closer approximations to the desired fixed point, and shows that the limit of these approximations exists, is a fixed point of the function under consideration, and is unique. The approximation process can be parameterized to some extent, and reused across multiple definitions that are "similar" enough. Furthermore these parameterized approximations can be composed hierarchically, yielding more powerful approximation techniques.

5.1 Definition

To make the notion of approximation precise, we need a way of stating how "close" two potential approximations are to each other. One approach would be to define a suitable metric space[12] and use the corresponding distance function, which returns either a rational or real number, given any two elements in the domain of the metric space. However, proving that a series of approximations converges to a limit point often requires reasoning about exponentiation and division over a theory of rationals or reals. An alternative way to measure "closeness", which we call *converging equivalence relations* (CER), instead only involves reasoning about well-founded sets, such as the set of natural numbers, or the set of finite lists. In many cases we can prove a unique fixed point exists by performing a simple induction over the natural numbers, something which all of the current HOL theorem provers support well.

A converging equivalence relation consists of:

- A type ρ, called the *resolution space*
- A type τ, called the *target space*
- A well-founded, transitive relation $(<)$ over type ρ, called a *resolution ordering*
- A three-argument predicate $(\approx)$ of type $(\rho \to \tau \to \tau \to bool)$, called an *indexed equivalence relation.* Given an element i of type ρ, and two elements x and y of type τ, we denote the application of $(\approx)$ to i, x and y as $(x \overset{i}{\approx} y)$, and if this value is true, then we say that x and y are *equivalent at resolution i.*

The resolution ordering $(<)$ and indexed equivalence relation $(\approx)$ must satisfy the properties in Fig. 5.1, for arbitrary $i, i' : \rho$; $x, y, z : \tau$; and $f : \rho \to \tau$. Axioms (5.1), (5.2), and (5.3) state that $(\approx)$ must be an equivalence relation at each resolution i. Axiom (5.4) states that if a resolution i has no lower resolutions, then $(\approx)$ treats all target elements as equivalent at that resolution. Such resolutions are called *minimal.* There is always at least one minimal resolution (and perhaps more than one), since $(<)$ is well-founded. Axiom (5.5) states that if two elements are equivalent at a particular resolution, then they are equivalent at all lower resolutions. Thus higher resolutions impose finer-grained, but compatible, partitions of the target space than lower resolutions do. Although no particular resolution may distinguish all elements, (5.6) states that if two elements are equivalent at all resolutions, then they are in fact equal.

Axioms (5.7) and (5.8) deal with "limits" of approximations. First some terminology: a function $f : \rho \to \tau$ from the space of resolutions to the target space of elements is called an *approximation map.* An approximation map f is *convergent up to resolution i* if for all resolutions j and j' such that $j < j' < i$, then $(f\, j)$ is equivalent at resolution j to $(f\, j')$. Note that it is possible for $(f\, i)$ itself not to be equivalent to any of the lower-resolution

$$x \overset{i}{\approx} x \tag{5.1}$$

$$x \overset{i}{\approx} y \longrightarrow y \overset{i}{\approx} x \tag{5.2}$$

$$x \overset{i}{\approx} y \wedge y \overset{i}{\approx} z \longrightarrow x \overset{i}{\approx} z \tag{5.3}$$

$$(\forall j \,.\, \neg(j < i)) \longrightarrow x \overset{i}{\approx} y \tag{5.4}$$

$$x \overset{i'}{\approx} y \wedge i < i' \longrightarrow x \overset{i}{\approx} y \tag{5.5}$$

$$(\forall j \,.\, x \overset{j}{\approx} y) \longrightarrow x = y \tag{5.6}$$

$$(\forall j, j' \,.\, j < j' < i \longrightarrow (f\, j) \overset{j}{\approx} (f\, j')) \longrightarrow (\exists z \,.\, \forall j < i \,.\, z \overset{j}{\approx} (f\, j)) \tag{5.7}$$

$$(\forall j, j' \,.\, j < j' \longrightarrow (f\, j) \overset{j}{\approx} (f\, j')) \longrightarrow (\exists z \,.\, \forall j \,.\, z \overset{j}{\approx} (f\, j)) \tag{5.8}$$

Figure 5.1: The CER axioms. Each of these axioms must hold for arbitrary i, x, y, and f.

$(f\, j)$'s. An approximation map f is *globally convergent* if for all resolutions j and j' such that $j < j'$, then $(f\, j) \overset{j}{\approx} (f\, j')$.

Axiom (5.7) states that if f is convergent up to resolution i, then there exists a limit-like element z that is equivalent at each resolution $j < i$ to the corresponding $(f\, j)$ approximation (there may be multiple such elements). Axiom (5.8) states that if f is globally convergent, then there exists a limit element z that is equivalent to each approximation $(f\, j)$ at resolution j.

5.2 Examples

5.2.1 Discrete CER

The simplest useful CER has as a resolution space a two-element type containing the values $\bot$ and $\top$, with $(\bot < \top)$, and a target space τ with $(\approx)$ defined such that $(x \overset{\bot}{\approx} y) \equiv \mathit{True}$, and $(x \overset{\top}{\approx} y) \equiv (x = y)$. Axioms (5.1) through (5.6) are easy to verify. Axiom (5.7) holds for any element. The limit element satisfying (5.8) is $f\, \top$.

5.2.2 Lazy list CER

We can construct a converging equivalence equation for comparing coinductive lists by comparing the first i elements of two lazy lists l_1 and l_2 at a given resolution i. To perform the comparison, we make use of the *ltake* function, with type $nat \to \alpha\, llist \to \alpha\, list$. The expression $(ltake\, n\, xs)$ returns a finite list consisting of the first n elements of xs. If xs has fewer than n elements, then *ltake* returns the whole of xs. The *ltake* function can be defined by well-founded recursion on its numeric argument with the following recursive equations:

$$\begin{array}{llllll} ltake & 0 & xs & = & [] \\ ltake & (n+1) & [] & = & [] \\ ltake & (n+1) & (x \# xs) & = & x \# (ltake\, n\, xs) \end{array}$$

We then define the lazy list CER with the natural numbers as the resolution space, $(\alpha\, llist)$ as the target space, the usual ordering on the natural numbers for $(<)$, and $(\approx)$ defined as follows:

$$xs \overset{i}{\approx} ys \equiv (ltake\, i\, xs = ltake\, i\, ys).$$

Axioms (5.1) through (5.3) hold trivially. The only minimal resolution in this CER is 0, and since $(ltake\, 0\, xs) = []$, then (5.4) holds. If two lazy lists are equal up to the first i positions, then they are equal up to any $i' < i$ position, so (5.5) holds. Axiom (5.6) reduces to the Take Lemma[75], which can be proved by coinduction.

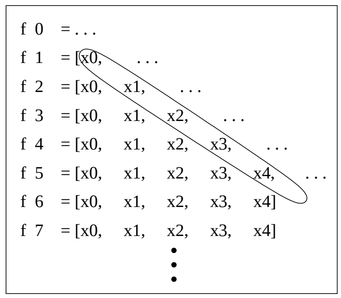
Axioms (5.7) and (5.8) require us to construct appropriate limit elements, given an approximation map. Both limit elements can be constructed by a single function, which we call *llist_diag*. For a given approximation map f, the limit elements may be of infinite length, so we define *llist_diag* by corecursion, using *llist_corec*:

$$llist\_diag \quad f \equiv llist\_corec\, 0\, (nthElem\, f)$$

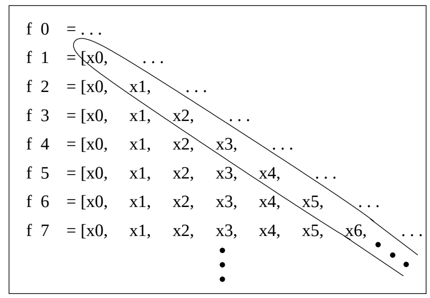
where

$$nthElem\, f\, n \equiv \begin{cases} \mathrm{Inl}\, (), & \text{if } ldrop\, n\, (f(n+1)) = [] \\ \mathrm{Inr}\, (x, n+1), & \text{if } ldrop\, n\, (f(n+1)) = (x \# xs) \end{cases}$$

The helper function *nthElem* uses the *ldrop* function on lazy lists. The *ldrop* function has type $nat \to (\alpha\, llist) \to (\alpha\, llist)$, and $(ldrop\, i\, xs)$ removes the first i elements from xs, returning the remainder. Like *ltake*, it is defined by well-founded recursion on its numeric



(a)



(b)

Figure 5.2: The *llist_diag* function constructs a limit list from an approximation map. In (a) the approximation map converges to a finite list; In (b) to an infinite list.

argument:

$$\begin{array}{llcll} \mathit{ldrop} & 0 & \mathit{xs} & = & \mathit{xs} \\ \mathit{ldrop} & (n+1) & [] & = & [] \\ \mathit{ldrop} & (n+1) & (x \mathbin{\#} \mathit{xs}) & = & \mathit{ldrop}\, n\, \mathit{xs} \end{array}$$

The overall action of *llist_diag* is to construct a so-called *diagonal list* from the approximation map f, where the n^{th} element of the result list is drawn from the n^{th} element of approximation $f\,(n+1)$, if the n^{th} element exists. If the n^{th} element does not exist (i.e., the length of $f\,(n+1)$ is less than n), then the result list is terminated at that point. This process is shown in Fig. 5.2. There are two possible cases. In Fig. 5.2-a, we see that the approximation map f converges to the finite list $[x_0,\ x_1,\ x_2,\ x_3,\ x_4]$. In Fig. 5.2-b, the approximation map f is converging to the infinite list $[x_0,\ x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6,\ \ldots]$

It turns out that for any CER whose $(<)$ relation is the less-than ordering on the natural numbers, the following property implies both (5.7) and (5.8):

$$\forall f\,.\,(\forall i\,.\,(f\ i) \overset{i}{\approx} (f\ (i+1))) \longrightarrow (\exists x\,.\,\forall i\,.\,x \overset{i}{\approx} (f\ i)).$$

With some work, one can show that this property holds for the lazy list CER by supplying *llist_diag* f as the existential witness element for x.

5.3 Contracting functions and the CER fixpoint theorem

In the theory of metric spaces, a *contracting function* is a function F such that for any two points x and y, $F\,x$ is closer to $F\,y$ than x is to y, given a suitable distance function.

Banach's theorem states that all contracting functions over suitable metric spaces have unique fixed points. We can define an analogous notion over a CER:

Definition 1 *A function F is* contracting *over a CER given by* $(<)$ *and* $(\approx)$ *if for all resolutions i and target elements x and y,*

$$(\forall i' < i \,.\, x \overset{i'}{\approx} y) \longrightarrow (F\,x) \overset{i}{\approx} (F\,y).$$

Intuitively, a function is contracting if, given two elements x and y that are close enough together at all lower resolutions $i' < i$ to satisfy the CER, but are potentially too far away at resolution i, then F maps them to two elements that are now close enough at resolution i.

For example, the function $\mathit{consZero}\ \mathit{xs} \equiv (0\#\mathit{xs})$ is contracting over the lazy list CER, since given any i and two lazy lists xs and ys,

$$(\forall i' < i \,.\, \mathit{ltake}\ i'\ \mathit{xs} = \mathit{ltake}\ i'\ \mathit{ys}) \longrightarrow \mathit{ltake}\ i\ (\mathit{consZero}\ \mathit{xs}) = \mathit{ltake}\ i\ (\mathit{consZero}\ \mathit{ys}).$$

The main result of this chapter is as follows:

CER Fixpoint Theorem *A contracting function F over a CER has a unique fixed point.*

The proof is discussed in Sect. 5.9. For now, we would like to apply this theorem to define some simple recursive functions over lazy lists.

5.4 Recursive definitions over coinductive lists

To begin with, we can simplify the definition of a contracting function F over a CER when the $(<)$ relation of that CER is the less-than relation over the natural numbers. In this case, Definition 1 reduces to

$$\forall\, i\, x\, y \,.\, x \overset{i}{\approx} y \longrightarrow (F\,x) \overset{i+1}{\approx} (F\,y). \tag{5.9}$$

Specializing this formula for the lazy list CER, we have that F is contracting on lazy lists if

$$\forall\, i\, x\, y \,.\, \mathit{ltake}\ i\ x = \mathit{ltake}\ i\ y \longrightarrow \mathit{ltake}\ (i+1)\ (F\,x) = \mathit{ltake}\ (i+1)\ (F\,y). \tag{5.10}$$

5.4.1 Defining *iterates*

Let us establish that the following recursive equation, defined over x and f, has a unique solution, and is thus a definition:

$$iterates = (x \# (lmap\ f\ iterates)) \tag{5.11}$$

This equation builds the infinite list $[x, f\,x, f\,(f\,x), \ldots]$. We first define the non-recursive functional F that characterizes this equation:

$$F\ iterates' \equiv (x \# (lmap\ f\ iterates')).$$

and then show that it is a contracting function. To do this we rely on (5.10), and assume we have two arbitrary lazy lists xs and ys such that $ltake\,i\,xs = ltake\,i\,ys$. We now need to show that $ltake\,(i+1)\,(F\,xs) = ltake\,(i+1)\,(F\,ys)$. Using a process of equational simplification we are able to reduce the goal to the assumption, as follows:

$$\begin{array}{ll} & ltake\ (i+1)\ (F\ xs) = ltake\ (i+1)\ (F\ ys) \\ \Leftrightarrow & ltake\ (i+1)\ (x \# (lmap\ f\ xs)) = ltake\ (i+1)\ (x \# (lmap\ f\ ys)) \\ \Leftrightarrow & ltake\ i\ (lmap\ f\ xs) = ltake\ i\ (lmap\ f\ ys) \\ \Leftarrow & ltake\ i\ xs = ltake\ i\ ys \end{array}$$

The simplification relies on the following facts, each proved by induction on i:

$$\begin{array}{rcl} (ltake\ (i+1)\ (z \# xs) = ltake\ (i+1)\ (z \# ys)) & \Leftrightarrow & (ltake\ i\ xs) = ltake\ i\ ys) \\ (ltake\ i\ (lmap\ f\ xs) = ltake\ i\ (lmap\ f\ ys) & \Leftarrow & (ltake\ i\ xs = ltake\ i\ ys) \end{array}$$

These facts illustrate a nice property of this proof: We did not have to expand the definitions of $(\#)$ or *lmap* during the simplification process, relying instead on an abstract characterization of their behavior with respect to *ltake*. This turns out to be the case for many functions, even recursive ones defined by contracting functions. In general we can often incrementally define recursive functions and prove properties about how they behave with respect to $(\approx)$, without having to expand the definitions of functions making up the body of the recursive definition.

5.5 Composing converging equivalence relations

The lazy list CER allows us to give recursive definitions of individual lazy lists, but we are often more interested in recursively defining functions that transform lazy lists. Fortunately, there are several *CER combinators* that allow us to build CERs over complex types, if we have CERs that operate on the corresponding atomic types.

Local and global limits

When constructing a new CER C' out of an existing CER C, we usually have to show (5.7) and (5.8) hold for C' by invoking (5.7) and (5.8) for C, to create the necessary limit witness elements. To make this process explicit, we use Hilbert's choice operator (ε) to create functions that return these witness elements[1], given an appropriate approximation mapping f:

$$\begin{aligned} \mathit{local\_limit} &\;::\; (\rho \to \tau) \to \rho \to \tau \\ \mathit{local\_limit}\, f\, i &\;\equiv\; (\varepsilon z \,.\, \forall j < i \,.\, z \overset{j}{\approx} (f\, j)) \end{aligned} \tag{5.12}$$

$$\begin{aligned} \mathit{global\_limit} &\;::\; (\rho \to \tau) \to \tau \\ \mathit{global\_limit}\, f &\;\equiv\; (\varepsilon z \,.\, \forall j \,.\, z \overset{j}{\approx} (f\, j)) \end{aligned} \tag{5.13}$$

We can use the axiom of choice for HOL, as well as (5.7) and (5.8) to prove the basic properties we want *local_limit* and *global_limit* to have for any CER given by ($<$) and ($\approx$):

$$(\forall j, j' \,.\, j < j' < i \longrightarrow (f\, j) \overset{j}{\approx} (f\, j')) \;\longrightarrow\; (\forall j < i \,.\, (\mathit{local\_limit}\, f\, i) \overset{j}{\approx} (f\, j)) \tag{5.14}$$

$$(\forall j, j' \,.\, j < j' \longrightarrow (f\, j) \overset{j}{\approx} (f\, j')) \;\longrightarrow\; (\forall j \,.\, (\mathit{global\_limit}\, f) \overset{j}{\approx} (f\, j)) \tag{5.15}$$

Function-space CER

The functions *local_limit* and *global_limit* allow us to concisely specify the limit elements of CER combinators. For example, given a CER C from resolution space ρ to target space

[1]This is merely a convenience. The CER properties can be shown with a little more work in Isabelle using (5.7) and (5.8) directly.

τ given by $(<)$ and $(\approx)$, we can construct a new *function-space over C* CER with the same resolution ordering $(<)$, and a new indexed equivalence relation $(\approx')$ with type $\rho \rightarrow (\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau) \rightarrow bool$, defined as

$$g \overset{i}{\approx}' h \equiv \forall x \,.\, (g\, x) \overset{i}{\approx} (h\, x).$$

The limit elements satisfying (5.7) and (5.8) can be given as

$$\begin{aligned} local\_limit\_fun\; f\; i &\equiv (\lambda x \,.\, local\_limit\; (\lambda i \,.\, f\; i\; x)\; i) \\ global\_limit\_fun\; f &\equiv (\lambda x \,.\, global\_limit\; (\lambda i \,.\, f\; i\; x)) \end{aligned}$$

Given these limit-producing functions, it is relatively easy to show that the function-space over C CER satisfies the CER axioms. As an example of the kind of reasoning involved, we prove that *local_limit_fun* satisfies (5.7).

Lemma 3 *Given a CER* $(<, \approx)$, *approximation map* f *of type* $\rho \rightarrow (\sigma \rightarrow \tau)$, *and* $(\approx')$ *defined as above, then if* $(\forall j, j' \,.\, j < j' < i \longrightarrow (f\; j) \overset{j}{\approx}' (f\; j'))$,
then $\forall j \,.\, local\_limit\_fun\; f\; i \overset{j}{\approx}' (f\; j)$.

Proof: Given the definition of $(\approx')$ and *local_limit_fun*, we must show for arbitrary x and j that $local\_limit\; f\; i\; x \overset{j}{\approx} f\; j\; x$. Let $f' \equiv \lambda i \,.\, f\; i\; x$. Then f' is an approximation map of type $\rho \rightarrow \tau$. Thus we need to show that $local\_limit\; f'\; i \overset{j}{\approx} f'\; j$. By definition of $(\approx')$ and the premise of the lemma, we have $(\forall j, j', x . j < j' < i \longrightarrow (f\; j\; x) \overset{j}{\approx} (f\; j'\; x))$. Applying the definition of f', we have $(\forall j, j' . j < j' < i \longrightarrow (f'\; j) \overset{j}{\approx} (f'\; j'))$. By (5.14) we have $\forall j < i. local\_limit\; f'\; i \overset{j}{\approx} (f'\; j)$, as desired. $\square$

5.5.1 Defining recursive functions with the function-space CER

Defining lmap

We can apply the function-space CER to define *lmap* recursively. The recursion equations for *lmap* are:

$$\begin{aligned} lmap\;\; f\;\; [] &= [] \\ lmap\;\; f\;\; (x\#xs) &= (f\; x)\,\#\,(lmap\; f\; xs) \end{aligned}$$

We translate the equations into a non-recursive form (parameterized over f)

$$\begin{array}{rll} F\ lmap' \equiv (\lambda xs\,. & \text{case}\ xs\ \text{of} & \\ & [] & \Rightarrow [] \\ & \mid (y \# ys) & \Rightarrow (f\ y) \# (lmap'\ ys)). \end{array}$$

We then need to show that $\text{fix}\, F$ is the unique fixed point of F by proving that F is a contracting function on the function-space over lazy lists CER. By (5.9) we must show for arbitrary resolution i and functions g and h, that $(g \overset{i}{\approx'} h \longrightarrow (F\ g) \overset{(i+1)}{\approx'} (F\ h))$. Expanding definitions, we obtain

$$\begin{array}{rl} & g \overset{i}{\approx'} h \longrightarrow (F\ g) \overset{(i+1)}{\approx'} (F\ h) \\ \Leftrightarrow & (\forall\, xs\,.\, g\ xs \overset{i}{\approx} h\ xs) \longrightarrow (\forall\, xs\,.\, (F\ g\ xs) \overset{(i+1)}{\approx} (F\ h\ xs)) \\ \Leftrightarrow & (\forall\, xs\,.\, ltake\ i\ (g\ xs) = ltake\ i\ (h\ xs)) \longrightarrow \\ & \quad (\forall\, xs\,.\, ltake\ (i+1)\ (F\ g\ xs) = ltake\ (i+1)\ (F\ h\ xs)). \end{array}$$

So, to prove F is contracting we take an arbitrary resolution i and two arbitrarily chosen functions g and h such that $(\forall\, xs\,.\, ltake\ i\ (g\ xs) = ltake\ i\ (h\ xs))$, and show for an arbitrary xs that $ltake\ (i+1)\ (F\ g\ xs) = ltake\ (i+1)\ (F\ h\ xs)$. There are two cases to consider:

case $xs = []$:

$$\begin{array}{rl} & ltake\ (i+1)\ (F\ g\ []) = ltake\ (i+1)\ (F\ h\ []) \\ \Leftrightarrow & ltake\ (i+1)\ [] = ltake\ (i+1)\ [] \\ \Leftrightarrow & \text{True}. \end{array}$$

case $xs = (y \# ys)$:

$$\begin{array}{rl} & ltake\ (i+1)\ (F\ g\ (y \# ys)) = ltake\ (i+1)\ (F\ h\ (y \# ys)) \\ \Leftrightarrow & ltake\ (i+1)\ ((f\ y) \# (g\ ys)) = ltake\ (i+1)\ ((f\ y) \# (h\ ys)) \\ \Leftrightarrow & ltake\ i\ (g\ ys) = ltake\ i\ (h\ ys) \\ \Leftrightarrow & \text{True}\ \{\text{by assumption}\}. \end{array}$$

Given the definition of F and basic lemmas about *ltake*, Isabelle's high-level simplification tactics allow the above proof to be carried out in two steps. The proof completes in about a second on a 266MHz Pentium II.

Defining lappend

We can apply the function-space CER combinator repeatedly, to prove that multi-argument curried functions have unique fixed points. As a concrete example, the curried function *lappend* has type $\alpha\, llist \to \alpha\, llist \to \alpha\, llist$. It takes two lazy list arguments xs and ys and returns a new list consisting of the elements of xs followed by the elements of ys. The recursive equations for *lappend* are

$$\begin{array}{llcll} lappend & [] & ys & = & ys \\ lappend & (x \# xs) & ys & = & (x \# lappend\ xs\ ys) \end{array}$$

To prove that these equations have a unique solution, we apply the function-space CER combinator to the lazy list CER to obtain a new CER C'. We then apply the function-space CER combinator again to C', obtaining a new CER C'' with the usual less-than relation on nat for $(<)$ and the following indexed equivalence relation $(\approx'')$:

$$g \overset{i}{\approx''} h \equiv (\forall\, xs\ ys\ .\ ltake\ i\ (g\ xs\ ys) = ltake\ i\ (h\ xs\ ys)).$$

Next, we convert the recursive equations for *lappend* into a non-recursive function F:

$$\begin{array}{ll} F\ lappend' \equiv (\lambda xs\ ys\ .\ \text{case } xs \text{ of} & \\ \qquad\qquad\qquad\qquad\quad [] & \Rightarrow ys \\ \qquad\qquad\qquad\qquad\ \mid\ (x \# xs') & \Rightarrow (x \# (lappend'\ xs'\ ys))). \end{array}$$

By (5.9) we must show for arbitrary resolution i and functions g and h, that

$$\begin{array}{l} (\forall\, xs\ ys\ .\ ltake\ i\ (g\ xs\ ys) = ltake\ i\ (h\ xs\ ys)) \longrightarrow \\ \quad (\forall\, xs\ ys\ .\ ltake\ (i+1)\ (F\ g\ xs\ ys) = ltake\ (i+1)\ (F\ h\ xs\ ys)). \end{array}$$

So we take arbitrary i, xs, and ys, and prove

$$ltake\ (i+1)\ (F\ g\ xs\ ys) = ltake\ (i+1)\ (F\ h\ xs\ ys)$$

assuming we have $(\forall\, xs\ ys\ .\ ltake\ i\ (g\ xs\ ys) = ltake\ i\ (h\ xs\ ys))$. There are two cases to consider, depending on whether xs is empty or not:

case $xs = []$:

$$
\begin{aligned}
& \mathit{ltake}\,(i+1)\,(F\,g\,[]\,ys) = \mathit{ltake}\,(i+1)\,(F\,h\,[]\,ys) \\
\Leftrightarrow\ & \mathit{ltake}\,(i+1)\,ys = \mathit{ltake}\,(i+1)\,ys \\
\Leftrightarrow\ & \text{True.}
\end{aligned}
$$

case $xs = (x\#xs')$:

$$
\begin{aligned}
& \mathit{ltake}\,(i+1)\,(F\,g\,(x\#xs')\,ys) = \mathit{ltake}\,(i+1)\,(F\,h\,(x\#xs')\,ys) \\
\Leftrightarrow\ & \mathit{ltake}\,(i+1)\,(x\,\#\,(g\,xs'\,ys)) = \mathit{ltake}\,(i+1)\,(x\,\#\,(h\,xs'\,ys)) \\
\Leftrightarrow\ & \mathit{ltake}\,i\,(g\,xs'\,ys) = \mathit{ltake}\,i\,(h\,xs'\,ys) \\
\Leftrightarrow\ & \text{True } \{\text{by assumption}\}.
\end{aligned}
$$

Thus we can conclude that *lappend* has a unique fixed point definition. We were able to carry out this proof in Isabelle in three steps, again taking about a second of CPU time.

5.5.2 Other CER combinators

CER combinators can also be defined over product and sum types. The lazy list CER can be generalized to work over any coinductive type that has a notion of depth, such as coinductive trees. A more powerful function-space CER is discussed in Sect. 5.7.

5.6 Demonstrating equality between coinductive elements

Converging equivalence relations can also be useful in showing that two elements of a target space are equal. Axiom (5.6) (restated below) says that to show two target elements x and y are equal, one simply needs to show they are equivalent at all resolutions j

$$(\forall j\,.\,x \overset{j}{\approx} y) \longrightarrow x = y.$$

We can often demonstrate that x and y are equivalent at all resolutions by well-founded induction, since $(<)$ is a well-founded relation. For example, given two arbitrary lazy lists ys and zs, we can prove the following lemma about *lappend*.

Lemma 4 $\forall xs\,.\,\mathit{ltake}\,i\,(\mathit{lappend}\,(\mathit{lappend}\,xs\,ys)\,zs) = \mathit{ltake}\,i\,(\mathit{lappend}\,xs\,(\mathit{lappend}\,ys\,zs))$.

Proof

case $i = 0$:

Take xs to be an arbitrary lazy list. Then

$$
\begin{aligned}
& \mathit{ltake}\, i\, (\mathit{lappend}\ (\mathit{lappend}\ \mathit{xs}\ \mathit{ys})\ \mathit{zs}) = \mathit{ltake}\, i\, (\mathit{lappend}\ \mathit{xs}\ (\mathit{lappend}\ \mathit{ys}\ \mathit{zs}))\\
\Leftrightarrow\ & \mathit{ltake}\, 0\, (\mathit{lappend}\ (\mathit{lappend}\ \mathit{xs}\ \mathit{ys})\ \mathit{zs}) = \mathit{ltake}\, 0\, (\mathit{lappend}\ \mathit{xs}\ (\mathit{lappend}\ \mathit{ys}\ \mathit{zs}))\\
\Leftrightarrow\ & [] = []\\
\Leftrightarrow\ & \text{True.}
\end{aligned}
$$

case $i = (k + 1)$:

Induction hypothesis:

$$
\begin{aligned}
\text{Assume } (\forall \mathit{xs}\, .\ & \mathit{ltake}\ k\ (\mathit{lappend}\ (\mathit{lappend}\ \mathit{xs}\ \mathit{ys})\ \mathit{zs}) =\\
& \mathit{ltake}\ k\ (\mathit{lappend}\ \mathit{xs}\ (\mathit{lappend}\ \mathit{ys}\ \mathit{zs})))
\end{aligned}
$$

Take xs to be an arbitrary lazy list. Then

$$
\begin{aligned}
& \mathit{ltake}\, i\, (\mathit{lappend}\ (\mathit{lappend}\ \mathit{xs}\ \mathit{ys})\ \mathit{zs}) = \mathit{ltake}\, i\, (\mathit{lappend}\ \mathit{xs}\ (\mathit{lappend}\ \mathit{ys}\ \mathit{zs}))\\
\Leftrightarrow\ & (\mathit{ltake}\ (k+1)\ (\mathit{lappend}\ (\mathit{lappend}\ \mathit{xs}\ \mathit{ys})\ \mathit{zs}) =\\
& \qquad \mathit{ltake}\ (k+1)\ (\mathit{lappend}\ \mathit{xs}\ (\mathit{lappend}\ \mathit{ys}\ \mathit{zs})))
\end{aligned}
$$

subcase $xs = []$:

$$
\begin{aligned}
\Leftrightarrow\ & (\mathit{ltake}\ (k+1)\ (\mathit{lappend}\ (\mathit{lappend}\ []\ \mathit{ys})\ \mathit{zs}) =\\
& \qquad \mathit{ltake}\ (k+1)\ (\mathit{lappend}\ []\ (\mathit{lappend}\ \mathit{ys}\ \mathit{zs})))\\
\Leftrightarrow\ & (\mathit{ltake}\ (k+1)\ (\mathit{lappend}\ \mathit{ys}\ \mathit{zs}) =\\
& \qquad \mathit{ltake}\ (k+1)\ (\mathit{lappend}\ \mathit{ys}\ \mathit{zs}))\\
\Leftrightarrow\ & \text{True.}
\end{aligned}
$$

subcase $xs = (x \,\#\, xs')$:

$$
\begin{aligned}
\Leftrightarrow\ & (\mathit{ltake}\ (k+1)\ (\mathit{lappend}\ (\mathit{lappend}\ (x \,\#\, \mathit{xs}')\ \mathit{ys})\ \mathit{zs}) =\\
& \qquad \mathit{ltake}\ (k+1)\ (\mathit{lappend}\ (x \,\#\, \mathit{xs}')\ (\mathit{lappend}\ \mathit{ys}\ \mathit{zs})))\\
\Leftrightarrow\ & (\mathit{ltake}\ (k+1)\ (\mathit{lappend}\ (x \,\#\, (\mathit{lappend}\ \mathit{xs}'\ \mathit{ys}))\ \mathit{zs}) =\\
& \qquad \mathit{ltake}\ (k+1)\ (x \,\#\, (\mathit{lappend}\ \mathit{xs}'\ (\mathit{lappend}\ \mathit{ys}\ \mathit{zs}))))
\end{aligned}
$$

$$\Leftrightarrow (\mathit{ltake}\,(k+1)\,(x \mathbin{\#} (\mathit{lappend}\,(\mathit{lappend}\,\mathit{xs}'\,\mathit{ys})\,\mathit{zs})) =$$
$$\mathit{ltake}\,(k+1)\,(x \mathbin{\#} (\mathit{lappend}\,\mathit{xs}'\,(\mathit{lappend}\,\mathit{ys}\,\mathit{zs}))))$$
$$\Leftrightarrow (\mathit{ltake}\,k\,(\mathit{lappend}\,(\mathit{lappend}\,\mathit{xs}'\,\mathit{ys})\,\mathit{zs}) =$$
$$\mathit{ltake}\,k\,(\mathit{lappend}\,\mathit{xs}'\,(\mathit{lappend}\,\mathit{ys}\,\mathit{zs})))$$
$$\Leftrightarrow \text{True } \{\text{by induction hypothesis}\}.$$

This proof took four steps in Isabelle, and relied on the following facts about *lappend*, each proved in two steps by expanding *lappend*'s recursive definition once and simplifying:

$$\begin{aligned} \mathit{lappend}\,[]\,\mathit{ys} &= \mathit{ys} \\ \mathit{lappend}\,(x\#\mathit{xs})\,\mathit{ys} &= x \mathbin{\#} (\mathit{lappend}\,\mathit{xs}\,\mathit{ys}) \end{aligned}$$

Given Lemma 4 and CER axiom (5.6) instantiated to the lazy list CER, we can then easily show in one Isabelle step that $\mathit{lappend}\,(\mathit{lappend}\,\mathit{xs}\,\mathit{ys})\,\mathit{zs} = \mathit{lappend}\,\mathit{xs}\,(\mathit{lappend}\,\mathit{ys}\,\mathit{zs})$.

5.7 Defining functions with unbounded look-ahead

The list-processing functions defined so far examine their arguments by performing at most one pattern match on a lazy list before producing an element of a result list. However, there is a class of functions that can examine a potentially infinite amount of their argument lists before deciding the next element to output. An example is the *lazy filter* function of type $(\alpha \to \mathit{bool}) \to \alpha\,\mathit{llist} \to \alpha\,\mathit{llist}$, which takes a predicate P and a lazy list xs, and returns a lazy list of the same type consisting only of those elements of xs satisfying P. A candidate set of recursion equations for this function might be

$$\begin{array}{lll} \mathit{lfilter}\,P\,[] & = [] & \\ \mathit{lfilter}\,P\,(x\#\mathit{xs}) & = \mathit{lfilter}\,P\,xs, & \text{if } \neg(P\,x) \\ \mathit{lfilter}\,P\,(x\#\mathit{xs}) & = x \mathbin{\#} (\mathit{lfilter}\,P\,xs), & \text{if } P\,x \end{array}$$

Sadly, this intuitively appealing set of equations does not completely define *lfilter*. If *lfilter* is given an infinite list xs, none of whose elements satisfy P, then the above equations do not specify what the result list should be. The *lfilter* function is free to return any value at all in this case. In other words, the equations do not have a unique solution.

Happily, however, we can remedy the situation as follows: We define by induction over *nat* a predicate *firstPelemAt* of type $(\alpha \to \mathit{bool}) \to \alpha\,\mathit{llist} \to \mathit{nat} \to \mathit{bool}$. The expression $(\mathit{firstPelemAt}\,P\,\mathit{xs}\,i)$ is true if *xs* has at least $(i+1)$ elements and i is the position of the first element of *xs* satisfying P. We can then define the predicate *never* of type $(\alpha \to \mathit{bool}) \to \alpha\,\mathit{llist} \to \mathit{bool}$ as

$$\mathit{never}\,P\,\mathit{xs} \equiv \forall\, i\,.\,\neg(\mathit{firstPelemAt}\,P\,\mathit{xs}\,i)$$

which is true when there are no elements in *xs* satisfying P. If we modify the initial recursive equations as follows:

$$\begin{array}{lll} \mathit{lfilter}\,P\,\mathit{xs} & = [], & \text{if } \mathit{never}\,P\,\mathit{xs} \\ \mathit{lfilter}\,P\,(x\#\mathit{xs}) & = \mathit{lfilter}\,P\,xs, & \text{if } \neg(\mathit{never}\,P\,\mathit{xs})\,\wedge\,\neg(P\,x) \\ \mathit{lfilter}\,P\,(x\#\mathit{xs}) & = x\,\#\,(\mathit{lfilter}\,P\,xs), & \text{if } P\,x \end{array}$$

then the set of equations does indeed have a unique solution. This function is not computable, since the predicate *never* can scan an infinite number of elements, but it is nevertheless mathematically valid in HOL. We can define a *well-founded function-space* CER combinator that is powerful enough to prove this. Given a CER C with $(<)$ of type $\rho \to \rho \to \mathit{bool}$ and $(\approx)$ with type $\rho \to \tau \to \tau \to \mathit{bool}$, and another well-founded transitive relation $(\prec)$ of type $\sigma \to \sigma \to \mathit{bool}$, we define our new CER C' with $(<')$ and $(\approx')$ as follows:

$$\begin{array}{l} (<') :: (\rho * \sigma) \to (\rho * \sigma) \to \mathit{bool} \\ (\approx') :: (\rho * \sigma) \to (\sigma \to \tau) \to (\sigma \to \tau) \to \mathit{bool} \end{array}$$

$$\begin{array}{ll} (a', t') <' (a, t) & \equiv a' < a \vee (a' = a \wedge t' \prec t) \\ g \overset{(a,t)}{\approx'} h & \equiv \forall\, a'\, t'\,.\,(a', t') \leq' (a, t) \longrightarrow (g\,t') \overset{a'}{\approx} (h\,t') \end{array}$$

It is a fair amount of work to show that C' is in fact a CER, so we elide the details.

Intuitively C' allows us to generalize well-founded recursion in the following way: A well-founded recursive function is forced to have its argument decrease in size on every recursive call. With C', the function being defined is allowed a choice; it can either decrease the size of its argument when making a recursive call, or not decrease its argument size but then make sure the element it is returning is "larger" than the element returned from its recursive call.

In the case of functions returning lazy lists, a "larger" lazy list is one that looks just like the lazy list returned by the recursive call, but with at least one extra element added to the front.

For us to use C' on *lfilter*, we need to specify a suitable well-founded transitive relation $(\prec)$. The relation we choose is one that holds when the first element satisfying P occurs sooner on the left-hand argument than on the right-hand argument:

$$xs \prec ys \equiv \mathit{firstPelem}\,P\,xs < \mathit{firstPelem}\,P\,ys$$

where

$$\begin{aligned} \mathit{firstPelem}\,P\,xs &= 0, && \text{if } \mathit{never}\,P\,xs \\ &= 1 + (\varepsilon i\,.\,\mathit{firstPelemAt}\,P\,xs\,i), && \text{otherwise} \end{aligned}$$

We arbitrarily decide that a list containing no P-elements is $\prec$-smaller than any list with at least one P-element.

When analyzing the revised recursive equations for *lfilter*, if xs has no P-elements then we return immediately, otherwise xs has to have at least one P-element. If that element is not at the head of the list, then the tail of the list is $\prec$-smaller than xs. If the first P-element is at the head of xs, then the tail of the list is not $\prec$-smaller than xs, but the output list has one more element than the list returned by the recursive call. Thus we informally conclude that *lfilter* is uniquely defined.

We have also proved this fact formally in Isabelle. After inductively proving various simple lemmas about *firstPelemAt*, *never*, and *firstPelem*, we were able to prove that *lfilter* is uniquely defined in five steps. We first translated the recursive equations above into a contracting function F. We used C' prove that F is contracting, first by expanding the definition of F and simplifying, and then by performing a case analysis (no induction required!) on whether the *nat* component of the current resolution was equal to zero. It took Isabelle two seconds to perform the proof.

Although we had to prove lemmas about *firstPelemAt*, *never*, and *firstPelem*, the proofs are not hard and it turns out we can reuse these results when defining other functions that perform unbounded search on lazy lists. For example, the *lflatten* function takes a lazy list of lazy lists, and flattens all of the elements into a single lazy list. The *lflatten* function can also be uniquely defined using *never*:

$$\begin{array}{lll} \mathit{lflatten}\ \mathit{xss} & = [], & \text{if } \mathit{never}\ (\lambda xs\ .xs \neq [])\ \mathit{xss} \\ \mathit{lflatten}\ (xs\#xss) & = \mathit{lappend}\ xs\ (\mathit{lflatten}\ xss), & \text{otherwise} \end{array}$$

The proof proceeds in Isabelle exactly as it does for *lfilter* except that we perform one additional case analysis on whether $xs = []$. The proof takes three seconds to complete.

5.8 Generalizing well-founded recursion

This section discusses how *WFFun*, the well-founded function space CER of Section 5.7 can be used to show that well-founded recursive function definitions have unique solutions.

WFFun is parameterized by two arguments: A well-founded relation $(\prec)$ and a base CER C. In Section 5.7, C was used to allow the function f being defined to call itself recursively on arguments that were not strictly $(\prec)$-smaller, provided that in this case f also returned a "larger" (i.e. more defined) result than the result of the recursive call. C was used to measure the definedness of the returned results.

In contrast, a well-founded function definition can call itself recursively on only strictly $(\prec)$-smaller arguments, but no requirements are placed on the function's return value. These requirements can be met in the CER framework by instantiating C to the discrete CER of Section 5.2.1. The discrete CER has only two resolutions, $\bot$ and $\top$, corresponding to completely undefined values and completely defined values, respectively.

To show that a fixed point functional F of type $(\rho \Rightarrow \tau) \Rightarrow (\rho \Rightarrow \tau)$ is contracting on the instantiated WFFun CER, it is sufficient to show that F satisfies the following formula for all i of type ρ and functions g and h of type $\rho \Rightarrow \tau$:

$$(\forall\ j.\ j \prec i \rightarrow g\ j = h\ j) \rightarrow F\ g\ i = F\ h\ i$$

In words, the formula states that when calling the recursive function at resolution i, the result only depends on recursive calls made at $(\prec)$-smaller values. That is, we can replace every recursive call in the body of the function being defined by a call to another function that only agrees with the "true" recursive function at arguments smaller than i, without changing the result of the overall expression. But this will be true if in fact the function is well-founded, since such functions only make recursive calls at smaller arguments.

From a theorem proving point of view, the formula above is particularly well suited to Isabelle's conditional rewriting tactics. In trying to show the formula holds for F, the rewriter will automatically convert the antecedent into a conditional rewrite rule, and then attempt to simplify the consequent. All applications of g in the left hand side of the consequent will be rewritten in terms of h by the added rewrite rule, provided the rewriter can show that g's argument is $(\prec)$-smaller than i. If it succeeds, then the left hand side will be syntactically equal to the right hand side, and the formula will simplify to the constant *True*.

5.9 Proof of the CER fixpoint theorem

5.9.1 Outline

Given a CER with resolution space ρ, target space τ, well-founded transitive relation $(<)$, indexed equivalence relation $(\approx)$, and an arbitrary contracting function F of type $\tau \to \tau$, our technique will be to construct an approximation map $\mathit{apx}\, F$ that converges globally to the desired fixed point. We then prove that this fixed point is unique by showing that any two fixed points of F are equal.

The function apx of type $(\tau \to \tau) \to \rho \to \tau$ that builds an approximation map from a contracting function is defined by well-founded recursion on $(<)$.

$$\mathit{apx}\, F\, i = F\, (\mathit{local\_limit}\, (\mathit{apx}\, F)\, i) \tag{5.16}$$

At each resolution i, the function apx uses $\mathit{local\_limit}$ to obtain the best possible approximation of $\mathrm{fix}\, F$, given the approximations it has already computed at all lower resolutions[2]. The result of calling *local_limit* may still not be close enough at resolution i, so *apx* maps the local limit through F, which will bring the result close enough. Isabelle's theory of well-founded functions ensures that the recursive instance of $apx\, F$ in the body of the definition is only applied to strictly smaller resolutions than i.

Once we have proved by well-founded induction that *apx* is well defined, we then establish that $apx\, F$ is convergent up to each resolution i, and that $apx\, F\, i \stackrel{i}{\approx} F\,(apx\, F\, i)$. This will allow us to show that $global\_limit\,(apx\, F) \stackrel{i}{\approx} F\,(global\_limit\,(apx\, F))$ at each resolution i, and are thus equal by (5.6). This result establishes that a fixed point exists for F. We then show that any two fixed points x and y of F are equivalent at all resolutions by well-founded induction, and thus are equal, again by (5.6).

5.9.2 Converging approximation maps

We assume throughout this treatment that $(<)$ and $(\approx)$ are arbitrary predicates satisfying the CER axioms, and that F is a contracting function over this CER. We do not bother to state these properties as premises of the lemmas and theorems below.

Our first task is to develop a theory of converging approximation maps, which will allow us to show in Section 5.9.3 that *apx* is globally convergent. To do this we need to define some terms.

Definition 2 *Two elements x and y of type τ are* equivalent up to resolution i *if $x \stackrel{j}{\approx} y$ for all $j < i$.*

Note that x and y do not have to be equivalent at resolution i itself to be equivalent up to resolution i.

Definition 3 *Given an element x of type τ and an approximation mapping f of type $\rho \rightarrow \tau$, then x* is a local limit at resolution i of f *if $x \stackrel{j}{\approx} (f\, j)$, for all $j < i$.*

Local limits imply local convergence:

Lemma 5 *If x is a local limit at resolution i of f, then f is convergent up to resolution i.*

[2]Here the definition of *local_limit* using Hilbert's choice operator seems essential.

Proof: Assuming arbitrary $k < j < i$, we must show $(f\,k) \stackrel{k}{\approx} (f\,j)$. Since x is a local limit at resolution i of f, then $(f\,j) \stackrel{j}{\approx} x$, and $(f\,k) \stackrel{k}{\approx} x$. Since $k < j$, then by (5.5) we have $(f\,j) \stackrel{k}{\approx} x$. Since $(\stackrel{k}{\approx})$ is an equivalence relation, then $(f\,k) \stackrel{k}{\approx} (f\,j)$ □

Lemma 6 *Given an approximation map f and resolution i, if for all $i' < i$ it is the case that $f\,i'$ is a local limit at resolution i' of f, then f is convergent up to resolution i.*

Proof: Assuming arbitrary $k < j < i$, we must show $(f\,k) \stackrel{k}{\approx} (f\,j)$. By assumption we have that $f\,j$ is a local limit at resolution j of f. That is, $\forall j' < j\,.\, f\,j \stackrel{j'}{\approx} f\,j'$. In particular, $f\,j \stackrel{k}{\approx} f\,k$, which is equal by (5.2) to $(f\,k) \stackrel{k}{\approx} (f\,j)$. □

Lemma 7 *If x and y are both local limits at resolution i of f, then x and y are equivalent up to resolution i.*

Proof: We must show for arbitrary $j < i$ that $x \stackrel{j}{\approx} y$. This holds since $x \stackrel{j}{\approx} (f\,j)$ and $y \stackrel{j}{\approx} (f\,j)$ by assumption, and since $(\stackrel{j}{\approx})$ is an equivalence relation. □

Lemma 8 *If f is locally convergent up to resolution i, then $local\_limit\,f\,i$ is a local limit at resolution i of f.*

Proof: By (5.7) we know there exists some element z such that $\forall j < i\,.\, z \stackrel{j}{\approx} (f\,j)$. By Definition 5.12 we have that $local\_limit\,f\,i = (\varepsilon z\,.\,\forall j < i\,.\, z \stackrel{j}{\approx} (f\,j))$. By the axiom of choice for HOL, we can conclude that $\forall j < i\,.\,(local\_limit\,f\,i) \stackrel{j}{\approx} (f\,j)$. That is, $local\_limit\,f\,i$ is a local limit at resolution i of f □

Lemma 9 *If f is globally convergent, then $global\_limit\,f$ is a global limit of f.*

Proof: By (5.8) we know there exists some element z such that $\forall i\,.\, z \stackrel{i}{\approx} (f\,i)$. By Definition 5.13 we have that $global\_limit\,f = (\varepsilon z\,.\,\forall i\,.\, z \stackrel{i}{\approx} (f\,i))$. By the axiom of choice for HOL, we can conclude that $\forall i\,.\, global\_limit\,f \stackrel{i}{\approx} (f\,i)$. That is, $global\_limit\,f$ is a global limit of f □

Lemma 10 *If* $x \overset{i}{\approx} y$*, and* G *is a contracting function, then* $G\,x \overset{i}{\approx} G\,y$.

Proof: If $x \overset{i}{\approx} y$, then x is equivalent to y at all lower resolutions, by (5.5). Thus x and y are equivalent up to resolution i. Thus by the definition of contracting function, $G\,x \overset{i}{\approx} G\,y$. $\square$

Lemma 11 *If* x *is a local limit at resolution* i *of* f*, and* G *is a contracting function, then* $G\,x$ *is a local limit at resolution* i *of* $G \circ f$.

Proof: Given arbitrary $j < i$, we must show that $G\,x \overset{j}{\approx} G\,(f\,j)$. By assumption we have $x \overset{j}{\approx} f\,j$. Then by Lemma 10 we have $G\,x \overset{j}{\approx} G\,(f\,j)$, as desired. $\square$

5.9.3 Properties of *apx*

Before we can establish that $apx\,F$ converges to the desired fixed point of F, we need to show that *apx* is a valid well-founded recursive definition. We will accomplish this using Isabelle's theory of well-founded relations, which contains a general recursion operator, *wfrec*, with type

$$(\alpha * \alpha)\;set \to ((\alpha \to \beta) \to (\alpha \to \beta)) \to \alpha \to \beta$$

The theory contains a theorem stating that if $(<)$ is a well-founded relation, then *wfrec* satisfies the following law:

$$wfrec\,(<)\,H\,a = H\,(cut\,(wfrec\,(<)\,H)\,a)\,a$$

where

$$cut\,f\,i\,x \equiv \text{if } x < i \text{ then } f\,x \text{ else } arbitrary$$

The helper function *cut* is used to ensure that recursive calls to $wfrec\,(<)\,H$ are only made at $(<)$-smaller values than a, ensuring well-foundedness. If H attempts to invoke $wfrec\,(<)\,H$ with any other value, then *cut* returns a fixed arbitrary element instead. We can then define *apx* as follows:

$$apx\ F\ i \equiv wfrec\ (<)\ H\ i$$

where

$$H\ apx'\ i \equiv F\ (local\_limit\ apx'\ i)$$

This non-recursive version of *apx* satisfies Isabelle's requirements for definitions. We now need to prove (5.16) as a lemma.

Lemma 12 *$apx\ F\ i = F\ (local\_limit\ (apx\ F)\ i)$*

Proof:

$apx\ F\ i$

$= \{\text{Def. of } apx\}$

$wfrec\ (<)\ H\ i$

$= \{wfrec \text{ law}\}$

$H\ (cut\ (wfrec\ (<)\ H)\ i)\ i$

$= \{\text{Def. of } apx \text{ in reverse}\}$

$H\ (cut\ (apx\ F)\ i)\ i$

$= \{\text{Def. of } H\}$

$F\ (local\_limit\ (cut\ (apx\ F)\ i)\ i)$

$= \{\text{Def. of } local\_limit\}$

$F\ (\varepsilon z\,.\,\forall j < i\,.\,z \overset{j}{\approx} ((cut\ (apx\ F)\ i)\ j))$

$= \{\text{Def. of } cut\text{, and } j < i \text{ in the body of the universal quantifier}\}$

$F\ (\varepsilon z\,.\,\forall j < i\,.\,z \overset{j}{\approx} (apx\ F\ j))$

$= \{\text{Def. of } local\_limit\}$

$F\ (local\_limit\ (apx\ F)\ i)$ $\square$

We now proceed to show that $apx\ F$ globally converges to the unique fixed point of F.

Lemma 13 *If $(apx\ F\ i)$ is a local limit at resolution i of $apx\ F$, then $apx\ F\ i \overset{i}{\approx} F\ (apx\ F\ i)$*

Proof: We have that $apx\ F$ is convergent up to resolution i by Lemma 5. By Lemma 8 $local\_limit\ (apx\ F)\ i$ is also a local limit at resolution i of $apx\ F$. Therefore $local\_limit\ (apx\ F)\ i$

and $apx\,F\,i$ are convergent up to resolution i. By the definition of contracting function, we have $F\,(local\_limit\,(apx\,F)\,i) \stackrel{i}{\approx} F\,(apx\,F\,i)$. By Lemma 12, this is equal to $apx\,F\,i \stackrel{i}{\approx} F\,(apx\,F\,i)$ $\square$

Lemma 14 *For all resolutions i, $apx\,F\,i$ is a local limit at resolution i to $apx\,F$.*

Proof: By well-founded induction on i. Thus we assume for all $j < i$ that $apx\,F\,j$ is a local limit at resolution j to $apx\,F$. By the induction hypothesis and Lemma 6 we have that $apx\,F$ is convergent up to resolution i. By Lemma 8 we have $local\_limit\,(apx\,F)\,i$ is a local limit at resolution i of $apx\,F$. By Lemma 11 we have that $F\,(local\_limit\,(apx\,F)\,i)$ is a local limit at resolution i of $F \circ apx\,F$. This means that $apx\,F\,i$ is a local limit at resolution i of $F \circ apx\,F$, by Lemma 12.

To show that $apx\,F\,i$ is a local limit at resolution i of $apx\,F$, we need to show for arbitrary $j < i$ that $apx\,F\,i \stackrel{j}{\approx} apx\,F\,j$. Since $apx\,F\,i$ is a local limit at resolution i of $F \circ apx\,F$, then $apx\,F\,i \stackrel{j}{\approx} F\,(apx\,F\,j)$. By the induction hypothesis and Lemma 13 we have that $apx\,F\,j \stackrel{j}{\approx} F\,(apx\,F\,j)$. Since $(\stackrel{j}{\approx})$ is an equivalence relation, we have $apx\,F\,i \stackrel{j}{\approx} apx\,F\,j$, as desired $\square$

Lemma 15 *For all resolutions i, $apx\,F\,i \stackrel{i}{\approx} F\,(apx\,F\,i)$*

Proof: By Lemmas 13 and 14 $\square$

Lemma 16 *$apx\,F$ is globally convergent.*

Proof: Given arbitrary i and j such that $i < j$, we must show that $apx\,F\,i \stackrel{i}{\approx} apx\,F\,j$. But this follows immediately from Lemma 14 and Definition 3 $\square$

Lemma 17 $global\_limit\,(apx\,F) = F\,(global\_limit\,(apx\,F))$.

Proof: Given an arbitrary resolution i, we have that $global\_limit\,(apx\,F) \stackrel{i}{\approx} apx\,F\,i$, by Lemma 9 and Lemma 16. We also have $F\,(global\_limit\,(apx\,F)) \stackrel{i}{\approx} F\,(apx\,F\,i)$, by Lemma 10. By Lemma 15 we have $apx\,F\,i \stackrel{i}{\approx} F\,(apx\,F\,i)$. Since $(\stackrel{i}{\approx})$ is an equivalence relation, we can conclude that $global\_limit\,(apx\,F) \stackrel{i}{\approx} F\,(global\_limit\,(apx\,F))$. Since i was arbitrarily chosen, the above equivalence holds for all resolutions i. Therefore the two values are equal, by (5.6) $\square$

This demonstrates that F has a fixed point. All that remains is to show that the fixed point is unique.

Lemma 18 *If $x = F\,x$ and $y = F\,y$ for contracting function F, then $x = y$.*

Proof: To show $x = y$ it suffices to show for arbitrary i that $x \stackrel{i}{\approx} y$, by (5.6). We shall demonstrate this by well-founded induction on i. Thus we assume that $x \stackrel{j}{\approx} y$, for all resolutions $j < i$. By the induction hypothesis and the definition of contracting function we have that $F\,x \stackrel{i}{\approx} F\,y$. Since $F\,x = x$ and $F\,y = y$, we conclude that $x \stackrel{i}{\approx} y$. $\square$

5.10 Applying CERs to Hawk circuits

The CER framework was originally developed to conservatively define recursive Hawk circuit definitions in higher order logic. Section 6.6.2 gives an example, where the internal state of a register file component is defined as a (higher order) recursive signal transformer called *envs*. Section 6.6.2 proves that *envs* uniquely satisfies its defining equation by creating a CER for signals and then demonstrating that the *envs* is contracting on the function space over signals CER.

5.11 Related work

The support for and application of well-founded induction and general coinduction has seen wide acceptance in the HOL theorem proving community. The well-founded definition package TFL used in HOL98 and Isabelle was written by Slind[88]. It can handle nested pattern matching in rule definitions, nested recursion in function bodies, and generates custom induction rules for each definition[87]. The PVS theorem prover[81] also uses well-founded induction as a basic definitional principle. A general theory of inductive and coinductive sets in Isabelle was developed by Paulson[75], based on least and greatest fixed points of monotone set-transforming functions, as well as a package for defining new inductive and coinductive sets by user-given introduction rules. The package avoids syntactic restrictions in the introduction rules by reasoning about each rule's underlying set-transformer semantics.

Paulson's Isabelle theories were applied by Frost[25] to formalize the static and dynamic semantics of a small functional language and prove that the two semantics were consistent with each other. Recursive functions are represented by infinitely nested environments, requiring consistency to be proved by coinduction. The underlying ideas of the language and proof, as well as the concept of coinduction as a variant of fixpoint induction, were introduced by Milner and Tofte[65].

A coinductive theory of streams (infinite-only lists) was developed by Miner[66] in the PVS theorem prover. Miner used this theory to model synchronous hardware circuits as corecursively-defined stream transformers. Using coinduction, he was able to optimize the implementation of a fault-tolerant clock synchronization circuit and a floating-point division circuit. In several cases a subcircuit was replaced by an optimized subcircuit, and the correctness of the replacement depended on non-trivial environmental assumptions in the surrounding circuit. Coinduction was used to verify the environmental assumptions and to show that the subcircuits were equivalent under the assumed environment.

A well-known alternative to coinductive types is the mathematical framework of *pointed complete partial orders* and *continuous functions*, also known as *domain theory*[32, 90]. This theory is supported by the HOLCF[68] object-logic in Isabelle, and also allows one to define infinite data structures such as lazy lists and trees. A wide variety of functions over these structures can then be recursively defined. The primary disadvantage of this approach is that one must add "extra" bottom-elements to the structures being defined. These extra elements are usually used to indicate non-termination. For example, a lazy filter function *lfilter* that removes all elements of a lazy list xs not satisfying a predicate P

can be defined recursively in HOLCF, but the expression *lfilter* $P\,xs$ returns $\perp$ instead of [] when xs is an infinite list containing no elements satisfying P. In contrast, Section 5.7 introduces a CER powerful enough to define an *lfilter* that returns [] in this case. Also, only so-called *admissible* predicates can be reasoned about inductively in domain theory, and it can be quite challenging to prove that a desired predicate is admissible. A comparison of the HOLCF approach to several other encodings of lazy lists is presented by Devillers et al[21].

Topology[12, 80] provides another well-established definition mechanism. The notions of Cauchy sequences, complete metric spaces, and contractions inspired much of this work. We have not worked out the exact relationship between converging equivalence relations and Cauchy metric spaces; although one can construct a distance function for every nat-indexed CER, it is not clear that distance functions can be always be constructed for more complex resolution spaces. Also, the conditions under which a function F is contracting in a CER seem to be less restrictive than the corresponding conditions in a metric space. More importantly from a verification perspective, well-founded induction seems easier to apply in current theorem provers than does the continuous mathematics required for metric spaces.

Chapter 6

Verifying the microarchitecture laws

Converging equivalence relations allow us to formally specify Hawk circuits as recursive equations over signals. We can use these equations to reason about Hawk components, and in particular prove the validity of the microarchitecture laws used in Chapter 3.

Many of the laws are localized enough that one can consider verifying them automatically by some kind of decision procedure. Since most decision procedures for hardware equivalence checking are based on state-machine transducer formalisms, a natural approach would be to first translate the left and right hand sides of the microarchitecture law being verified into state machine transducers, and then verify that the two transducers are observationally equivalent. Algorithms for performing such equivalency verifications on finite state machines have been extensively studied, including techniques based on Binary Decision Diagrams[17] and Stålmarck's Method[50]. In fact, several commercial tools now exist for performing equivalency checking on large hardware circuits.

These techniques cannot immediately be used on Hawk circuits, since the lack of a priori bounds on the size of words or the number of registers used in Hawk microarchitectures means that typical Hawk components translate into infinite state, instead of finite state, transducers. Fortunately, significant progress has also been made on checking the equivalence of infinite state machine transducers, using symmetry reduction[15, 24] and abstraction[16] techniques. Usually these techniques require some manual intervention, although often less than that required for pure theorem proving-based approaches.

However, in keeping with our theme of exploring algebraic methods for performing microarchitecture verification, we have chosen to continue verifying the individual laws themselves using a combination of equational reasoning and induction.

The equivalence proofs themselves can be quite large, even given the relatively simple component definitions needed to specify the pipeline of Chapter 3. It is not that the proofs are mathematically sophisticated, but rather that the components process large amounts of disparate data, namely the field values of transactions. The aim of this chapter is to give a flavor of the kind and amount of reasoning involved in proving two transaction-processing components behaviorally equivalent, and to present some techniques for reducing the size of the associated proof. Since even the "reduced" proofs of these laws can be quite lengthy we only sketch a couple of examples in this chapter: the *alu* time invariance law and the registerFile-bypass law.

6.1 A theory of transactions

The main source of proof complexity results from the large number of fields that a transaction contains, and the fact that the field values are of different types. A typical equivalence proof of two transaction-processing components F and G will involve a series of cases, one for each transaction field, showing that the two circuits output identical field values. Many of the cases will be symmetric with respect to each other, differing only in the name of the field mentioned in the proof and the field's type. To reduce the amount of redundancy in such proofs this section will present a *theory of transactions* where transaction fields themselves are logical objects, and can be quantified over. In this way a symmetric group of cases in a proof can be reduced to a single proof parameterized over the symmetrically-used field names.

We begin by precisely defining what a transaction is in higher order logic. Intuitively a transaction is a record containing all of the fields that a microarchitecture uses to process one instruction. The set of fields needed depends on the instruction set architecture and the complexity of the microarchitecture implementing it. For the branch-predicting microarchitecture we consider in this thesis, we require the following fields:

- **destRegFld :: Reg**

  The destination register name.

- **destValFld :: Word**

The destination register contents.

- **opcodeFld :: Opcode**

  The operation the transaction is to perform.

- **s1RegFld :: Reg**

  The first source operand register name.

- **s1ValFld :: Word**

  The first source operand register contents.

- **s2RegFld :: Reg**

  The second source operand register name.

- **s1ValFld :: Word**

  The second source operand register contents.

- **specPCFld :: Word**

  The speculative next address to fetch. This value is set by the branch target prediction buffer in the instruction cache. If the ALU calculates the actual next address for a branch instruction to be different from the speculative next address, then a branch misprediction has occurred.

- **nextPCFld :: Word**

  The actual next address to fetch. Initialized by the instruction cache to the address following the address the transaction was fetched from. On branch instructions, the ALU will set this field to the actual branch target address.

6.1.1 Transaction as an abstract datatype

There are many different ways in higher order logic to create such records. Rather than fix a particular model, we define a new type called *Trans*, with a function for constructing a transaction given initial values for each field, and a series of accessor functions, one for each transaction field.

$$
\begin{array}{ll}
\mathit{mkTrans}' & :: \mathit{Reg} \Rightarrow \mathit{Word} \Rightarrow \mathit{Opcode} \Rightarrow \mathit{Reg} \Rightarrow \mathit{Word} \Rightarrow \\
& \quad \mathit{Reg} \Rightarrow \mathit{Word} \Rightarrow \mathit{Word} \Rightarrow \mathit{Word} \Rightarrow \mathit{Trans} \\
\mathit{dstReg}' & :: \mathit{Trans} \Rightarrow \mathit{Reg} \\
\mathit{dstValName}' & :: \mathit{Trans} \Rightarrow \mathit{Word} \\
\mathit{opcode}' & :: \mathit{Trans} \Rightarrow \mathit{Opcode} \\
s1\mathit{Reg}' & :: \mathit{Trans} \Rightarrow \mathit{Reg} \\
s1\mathit{Val}' & :: \mathit{Trans} \Rightarrow \mathit{Word} \\
s2\mathit{Reg}' & :: \mathit{Trans} \Rightarrow \mathit{Reg} \\
s2\mathit{Val}' & :: \mathit{Trans} \Rightarrow \mathit{Word} \\
\mathit{specPC}' & :: \mathit{Trans} \Rightarrow \mathit{Word} \\
\mathit{nextPC}' & :: \mathit{Trans} \Rightarrow \mathit{Word}
\end{array}
$$

We will follow the convention that functions that take or return transactions or transaction fields will have a ($'$) appended to their name (as opposed to functions that operate on signals of transactions).

6.1.2 Transaction laws

There are two properties we want elements of the transaction type to satisfy. First, it must be the case that each field accessor function retrieves the same value as was used to construct that field of the transaction:

$$
\begin{array}{l}
\mathit{dstReg}'\ (\mathit{mkTrans}'\ \mathit{dstReg}\ \mathit{dstValName}\ \mathit{opc}\ s1\mathit{Reg} \\
\qquad\qquad s1\mathit{Val}\ s2\mathit{Reg}\ s2\mathit{Val}\ \mathit{specPC}\ \mathit{nextPC}) \ = \ \mathit{dstReg} \\
\mathit{dstValName}'\ (\mathit{mkTrans}'\ \mathit{dstReg}\ \mathit{dstValName}\ \mathit{opc}\ s1\mathit{Reg} \\
\qquad\qquad s1\mathit{Val}\ s2\mathit{Reg}\ s2\mathit{Val}\ \mathit{specPC}\ \mathit{nextPC}) \ = \ \mathit{dstValName} \\
\vdots \\
\mathit{nextPC}'\ (\mathit{mkTrans}'\ \mathit{dstReg}\ \mathit{dstValName}\ \mathit{opc}\ s1\mathit{Reg} \\
\qquad\qquad s1\mathit{Val}\ s2\mathit{Reg}\ s2\mathit{Val}\ \mathit{specPC}\ \mathit{nextPC}) \ = \ \mathit{nextPC}
\end{array}
$$

Second, it must be the case that two transactions are equal exactly when all of their fields are equal:

$$
\begin{aligned}
(tr1 = tr2) = (&dstReg'\ tr1 = dstReg'\ tr2 &&\wedge\\
&dstValName'\ tr1 = dstValName'\ tr2 &&\wedge\\
&opcode'\ tr1 = opcode'\ tr2 &&\wedge\\
&s1Reg'\ tr1 = s1Reg'\ tr2 &&\wedge\\
&s1Val'\ tr1 = s1Val'\ tr2 &&\wedge\\
&s2Reg'\ tr1 = s2Reg'\ tr2 &&\wedge\\
&s2Val'\ tr1 = s2Val'\ tr2 &&\wedge\\
&specPC'\ tr1 = specPC'\ tr2 &&\wedge\\
&nextPC'\ tr1 = nextPC'\ tr2)
\end{aligned}
$$

To prevent the possibility of logical inconsistencies, we use Isabelle's type definition package to define *Trans* and derive the appropriate laws as theorems. In our definitions, we define a transaction simply as a tuple of its fields, $mkTrans'$ as a function that constructs a tuple from its field arguments, and the field accessors as the appropriate tuple projections. Another choice would have been to use Isabelle's datatype package.

From the two transaction properties above we can show that $mkTrans'$ can construct any valid transaction tr by using the transaction accessors on tr itself.

$$
\begin{aligned}
(tr = mkTrans'\ &(dstReg'\ tr)\ (dstValName'\ tr)\ (opcode'\ tr)\\
&(s1Reg'\ tr)\ (s1Val'\ tr)\ (s2Reg'\ tr)\ (s2Val'\ tr)\\
&(specPC'\ tr)\ (nextPC'\ tr))
\end{aligned}
$$

$$= \{\text{second transaction property; use let expression to share common subterms}\}$$

$$
\begin{aligned}
(let\ tr2 = mkTrans'\ &(dstReg'\ tr)\ (dstValName'\ tr)\ (opcode'\ tr)\\
&(s1Reg'\ tr)\ (s1Val'\ tr)\ (s2Reg'\ tr)\ (s2Val'\ tr)\\
&(specPC'\ tr)\ (nextPC'\ tr)
\end{aligned}
$$

$$\begin{aligned}
&\mathit{in}\ \mathit{dstReg}'\ \mathit{tr}\ =\ \mathit{dstReg}'\ \mathit{tr}2 \wedge \\
&\quad \mathit{dstValName}'\ \mathit{tr}\ =\ \mathit{dstValName}'\ \mathit{tr}2 \wedge \\
&\quad \mathit{opcode}'\ \mathit{tr}\ =\ \mathit{opcode}'\ \mathit{tr}2 \wedge \\
&\quad \mathit{s1Reg}'\ \mathit{tr}\ =\ \mathit{s1Reg}'\ \mathit{tr}2 \wedge \\
&\quad \mathit{s1Val}'\ \mathit{tr}\ =\ \mathit{s1Val}'\ \mathit{tr}2 \wedge \\
&\quad \mathit{s2Reg}'\ \mathit{tr}\ =\ \mathit{s2Reg}'\ \mathit{tr}2 \wedge \\
&\quad \mathit{specPC}'\ \mathit{tr}\ =\ \mathit{specPC}'\ \mathit{tr}2 \wedge \\
&\quad \mathit{nextPC}'\ \mathit{tr}\ =\ \mathit{nextPC}'\ \mathit{tr}2)
\end{aligned}$$

$$= \{\text{expand let expression; first transaction property}\}$$

$$\begin{aligned}
&(\mathit{dstReg}'\ \mathit{tr}\ =\ \mathit{dstReg}'\ \mathit{tr} \wedge \\
&\ \mathit{dstValName}'\ \mathit{tr}\ =\ \mathit{dstValName}'\ \mathit{tr} \wedge \\
&\ \mathit{opcode}'\ \mathit{tr}\ =\ \mathit{opcode}'\ \mathit{tr} \wedge \\
&\ \mathit{s1Reg}'\ \mathit{tr}\ =\ \mathit{s1Reg}'\ \mathit{tr} \wedge \\
&\ \mathit{s1Val}'\ \mathit{tr}\ =\ \mathit{s1Val}'\ \mathit{tr} \wedge \\
&\ \mathit{s2Reg}'\ \mathit{tr}\ =\ \mathit{s2Reg}'\ \mathit{tr} \wedge \\
&\ \mathit{specPC}'\ \mathit{tr}\ =\ \mathit{specPC}'\ \mathit{tr} \wedge \\
&\ \mathit{nextPC}'\ \mathit{tr}\ =\ \mathit{nextPC}'\ \mathit{tr})
\end{aligned}$$

$$= \{\text{logic}\}$$

$$\mathit{True}$$

Thus we know that transactions contain no "hidden" fields.

6.1.3 Derived transaction operators

Many of the Hawk components take existing transactions and construct new transactions from them that change just a few fields. We can simplify the definitions of these components by defining a series of transaction *updaters*, each of which takes a transaction field value and an existing transaction and returns a new transaction just like the original except with the appropriate field updated:

$$\mathit{setDstReg}'\ ::\ \mathit{Reg} \Rightarrow \mathit{Trans} \Rightarrow \mathit{Trans}$$

$$\mathit{setDstReg}'\ \mathit{reg}\ \mathit{tr}\ =$$

$$\quad mkTrans'\ reg\ (dstValName'\ tr)\ (opcode'\ tr)\ (s1Reg'\ tr)\ (s1Val'\ tr)$$
$$\qquad (s2Reg'\ tr)\ (s2Val'\ tr)\ (specPC'\ tr)\ (nextPC'\ tr)$$

$$setDstVal' :: Word \Rightarrow Trans \Rightarrow Trans$$
$$setDstVal'\ val\ tr\ =$$
$$\quad mkTrans'\ (dstReg'\ tr)\ val\ (opcode'\ tr)\ (s1Reg'\ tr)\ (s1Val'\ tr)$$
$$\qquad (s2Reg'\ tr)\ (s2Val'\ tr)\ (specPC'\ tr)\ (nextPC'\ tr)$$
$$\vdots$$
$$setNextPC' :: Word \Rightarrow Trans \Rightarrow Trans$$
$$setNextPC'\ pc\ tr\ =$$
$$\quad mkTrans'\ (dstReg'\ tr)\ (dstValName'\ tr)\ (opcode'\ tr)\ (s1Reg'\ tr)\ (s1Val'\ tr)$$
$$\qquad (s2Reg'\ tr)\ (s2Val'\ tr)\ (specPC'\ tr)\ pc$$

We can derive several useful laws for these functions. Taking the $setDstReg'$ function as a representative example, we can show for an arbitrary transaction tr that the updater does in fact update the appropriate field:

$$dstReg'\ (setDstReg'\ reg\ tr)$$
$$\quad = \{\text{definition of } setDstReg'\}$$
$$dstReg'\ (mkTrans'\ reg\ (dstValName'\ tr)\ (opcode'\ tr)\ (s1Reg'\ tr)\ (s1Val'\ tr)$$
$$\qquad (s2Reg'\ tr)\ (s2Val'\ tr)\ (specPC'\ tr)\ (nextPC'\ tr))$$
$$\quad = \{\text{first transaction property}\}$$
$$reg$$

It is also the case that none of the other fields are modified, for example the opcode field:

$$opcode'\ (setDstReg'\ reg\ tr)$$
$$\quad = \{\text{definition of } setDstReg'\}$$
$$opcode'\ (mkTrans'\ reg\ (dstValName'\ tr)\ (opcode'\ tr)\ (s1Reg'\ tr)\ (s1Val'\ tr)$$
$$\qquad (s2Reg'\ tr)\ (s2Val'\ tr)\ (specPC'\ tr)\ (nextPC'\ tr))$$
$$\quad = \{\text{first transaction property}\}$$
$$opcode'\ tr$$

We can similarly show that all of the other transaction fields remain unchanged.

6.2 Exploiting symmetry in transaction fields

We would like to prove this last property as a general theorem. We can define (outside of higher order logic) the set of field accessors

$$A \equiv \{dstReg', dstValName', opcode', \ldots\}$$

and the set of field updaters

$$U \equiv \{setDstReg', setDstVal', opcode', \ldots\}$$

and define a bijection $update : A \rightarrow U$ that maps each field accessor to its corresponding field updater. Thus $update(dstReg') = setDstReg'$, $update(dstValName') = setDstVal'$, and so on. We would like to prove the following fact in higher order logic:

$$\begin{aligned} &\forall\, t \in Trans,\ fld \in A,\ fld' \in A,\ x \in dom(fld'). \\ &\quad fld \neq fld' \rightarrow fld\ (update(fld')\ x\ t) \ = \ fld\ t \end{aligned}$$

That is, if we update a field of a transaction and then examine a different field of the result, it should be the same as the original transaction's field. While we can prove that every instance of the above formula is true, the type system of higher order logic is too restrictive to allow us to prove the formula itself as a theorem. We cannot even construct the set A in higher order logic, since the elements of A are of different types.

Since any given microarchitectural component only modifies a few transaction fields, it would be nice if we could prove something like the above statement as a theorem and avoid having to re-prove that each of the other fields of the transaction returned by the component is unchanged. For example, the *alu* component only modifies the *dstValName* and *nextPC* fields. We would like to prove the following formula

$$\forall\, t \in Trans, n \in Time,\ fld \in A \ - \ \{dstValName',\ nextPC'\}.\ fld\ (alu\ t\ n) \ = \ fld\ (t\ n)$$

but we run into similar problems. As it stands we have to instead prove each instance of this formula as a separate theorem.

6.2.1 First class field names

We can work around HOL's inability to quantify over types by using a well-known technique from the typed functional programming community.

Instead of trying to define the set A of transaction accessors directly, we will define a new datatype of accessor *names*, called *FieldNm*, all of whose elements have the same type:

$$\mathit{datatype}\ \mathit{Operand}\ = \mathit{Dst} \mid \mathit{Src}1 \mid \mathit{Src}2$$

$$\begin{aligned}\mathit{datatype}\ \mathit{FieldNm}\ =\ & \mathit{RegNm}\ \mathit{Operand} \mid \mathit{ValNm}\ \mathit{Operand} \mid \\ & \mathit{opcodeNm} \mid \mathit{specPCNm} \mid \mathit{nextPCNm}\end{aligned}$$

Note that the *RegNm* and *ValNm* constructors have been parameterized by their operand location. Thus, for example, *RegNm Dst* is the name of the destination register field, and *ValNm* $\mathit{Src}1$ is the name of the field holding the first source operand register contents. We will also define a uniform datatype for holding the contents of a field:

$$\mathit{datatype}\ \mathit{FieldValue}\ =\ \mathit{RegValue}'\ \mathit{Reg} \mid \mathit{WordValue}'\ \mathit{Word} \mid \mathit{OpcodeValue}'\ \mathit{Opcode}$$

We can now create a single parameterized field accessor function that takes a field name and a transaction and returns the appropriate field contents as a *FieldValue*.

$$\begin{aligned}&\mathit{field}' :: \mathit{FieldNm} \Rightarrow \mathit{Trans} \Rightarrow \mathit{FieldValue}\\ &\mathit{field}'\ \mathit{nm}\ t\ =\end{aligned}$$

$$
\begin{array}{l}
\mathit{case}\ \mathit{nm}\ \mathit{of} \\
\quad \begin{array}{lcl}
\ \ (\mathit{RegNm}\ \mathit{Dst}) & \Rightarrow & \mathit{RegValue}'\ (\mathit{dstReg}'\ t) \\
\mid (\mathit{ValNm}\ \mathit{Dst}) & \Rightarrow & \mathit{WordValue}'\ (\mathit{dstValName}'\ t) \\
\mid \mathit{opcodeNm} & \Rightarrow & \mathit{OpcodeValue}'\ (\mathit{opcode}'\ t) \\
\mid (\mathit{RegNm}\ \mathit{Src}1) & \Rightarrow & \mathit{RegValue}'\ (s1\mathit{Reg}'\ t) \\
\mid (\mathit{ValNm}\ \mathit{Src}1) & \Rightarrow & \mathit{WordValue}'\ (s1\mathit{Val}'\ t) \\
\mid (\mathit{RegNm}\ \mathit{Src}2) & \Rightarrow & \mathit{RegValue}'\ (s2\mathit{Reg}'\ t) \\
\mid (\mathit{ValNm}\ \mathit{Src}2) & \Rightarrow & \mathit{WordValue}'\ (s2\mathit{Val}'\ t) \\
\mid \mathit{specPCNm} & \Rightarrow & \mathit{WordValue}'\ (\mathit{specPC}'\ t) \\
\mid \mathit{nextPCNm} & \Rightarrow & \mathit{WordValue}'\ (\mathit{nextPC}'\ t)
\end{array} \\
\mathit{end}
\end{array}
$$

We would like to create a parameterized field updater function in a similar fashion

$$\mathit{update}' :: \mathit{FieldNm} \Rightarrow \mathit{FieldValue} \Rightarrow \mathit{Trans} \Rightarrow \mathit{Trans}$$

but the primitive field updaters do not take *FieldValue* elements as parameters. To solve this problem we define a series of *type cast* functions, one for each constructor in *FieldValue*. We use Hilbert's choice operator to perform the cast. If the casting functions are given a *FieldValue* element that does not correspond to the type they are casting to, then the choice operator will return an arbitrary element of the correct type.

$$
\begin{array}{l}
\mathit{castToReg}' :: \mathit{FieldValue} \Rightarrow \mathit{Reg} \\
\mathit{castToReg}'\ \mathit{fv}\ =\ (\varepsilon\ r.\ \mathit{fv}\ =\ \mathit{RegValue}'\ r) \\
\\
\mathit{castToWord}' :: \mathit{FieldValue} \Rightarrow \mathit{Word} \\
\mathit{castToWord}'\ \mathit{fv}\ =\ (\varepsilon\ w.\ \mathit{fv}\ =\ \mathit{WordValue}'\ w) \\
\\
\mathit{castToOpcode}' :: \mathit{FieldValue} \Rightarrow \mathit{Opcode} \\
\mathit{castToOpcode}'\ \mathit{fv}\ =\ (\varepsilon\ \mathit{opc}.\ \mathit{fv}\ =\ \mathit{OpcodeValue}'\ \mathit{opc})
\end{array}
$$

We also define a predicate indicating whether a *FieldValue* element is compatible with a given *FieldNm*:

$$
\begin{array}{l}
\mathit{validFieldType} \;::\; \mathit{FieldNm} \Rightarrow \mathit{FieldValue} \Rightarrow \mathit{bool} \\
\mathit{validFieldType}\ \mathit{nm}\ \mathit{fv} \;= \\
\quad \mathit{case}\ \mathit{fv}\ \mathit{of} \\
\qquad (\mathit{RegValue}'\ r) \;\Rightarrow \quad (\mathit{nm} \;=\; (\mathit{RegNm}\ \mathit{Dst}) \vee \\
\qquad\qquad\qquad\qquad\quad\;\; \mathit{nm} \;=\; (\mathit{RegNm}\ \mathit{Src}1) \vee \\
\qquad\qquad\qquad\qquad\quad\;\; \mathit{nm} \;=\; (\mathit{RegNm}\ \mathit{Src}2)) \\
\qquad \mid (\mathit{WordValue}'\ w) \;\Rightarrow \quad (\mathit{nm} \;=\; (\mathit{ValNm}\ \mathit{Dst}) \vee \\
\qquad\qquad\qquad\qquad\quad\;\; \mathit{nm} \;=\; (\mathit{ValNm}\ \mathit{Src}1) \vee \\
\qquad\qquad\qquad\qquad\quad\;\; \mathit{nm} \;=\; (\mathit{ValNm}\ \mathit{Src}2) \vee \\
\qquad\qquad\qquad\qquad\quad\;\; \mathit{nm} \;=\; \mathit{specPCNm} \vee \\
\qquad\qquad\qquad\qquad\quad\;\; \mathit{nm} \;=\; \mathit{nextPCNm}) \\
\qquad \mid (\mathit{OpcodeValue}'\ \mathit{opc}) \;\Rightarrow\; \mathit{nm} \;=\; \mathit{opcodeNm} \\
\quad \mathit{end}
\end{array}
$$

We use the casting functions to define the parameterized field updater:

$$
\begin{array}{l}
\mathit{update}' \;::\; \mathit{FieldNm} \Rightarrow \mathit{FieldValue} \Rightarrow \mathit{Trans} \Rightarrow \mathit{Trans} \\
\mathit{update}'\ \mathit{nm}\ v \;= \\
\quad \mathit{case}\ \mathit{nm}\ \mathit{of} \\
\qquad (\mathit{RegNm}\ \mathit{Dst}) \;\Rightarrow\; \mathit{setDstReg}'\ (\mathit{castToReg}'\ v) \\
\qquad \mid (\mathit{ValNm}\ \mathit{Dst}) \;\Rightarrow\; \mathit{setDstVal}'\ (\mathit{castToWord}'\ v) \\
\qquad \mid \mathit{opcodeNm} \;\Rightarrow\; \mathit{setOpcode}'\ (\mathit{castToOpcode}'\ v) \\
\qquad \mid (\mathit{RegNm}\ \mathit{Src}1) \;\Rightarrow\; \mathit{setS1Reg}'\ (\mathit{castToReg}'\ v) \\
\qquad \mid (\mathit{ValNm}\ \mathit{Src}1) \;\Rightarrow\; \mathit{setS1Val}'\ (\mathit{castToWord}'\ v) \\
\qquad \mid (\mathit{RegNm}\ \mathit{Src}2) \;\Rightarrow\; \mathit{setS2Reg}'\ (\mathit{castToReg}'\ v) \\
\qquad \mid (\mathit{ValNm}\ \mathit{Src}2) \;\Rightarrow\; \mathit{setS2Val}'\ (\mathit{castToWord}'\ v) \\
\qquad \mid \mathit{specPCNm} \;\Rightarrow\; \mathit{setSpecPC}'\ (\mathit{castToWord}'\ v) \\
\qquad \mid \mathit{nextPCNm} \;\Rightarrow\; \mathit{setNextPC}'\ (\mathit{castToWord}'\ v) \\
\quad \mathit{end}
\end{array}
$$

6.2.2 Generalized field laws

Variants of the previous formulas (that couldn't be stated in higher order logic) can now be proved as theorems

$$\begin{aligned}
&\mathit{validFieldType}\ \mathit{nm}\ x\ \rightarrow\ \mathit{field}'\ \mathit{nm}\ (\mathit{update}'\ \mathit{nm}\ x\ t)\ =\ x\\
&\mathit{nm}\ \neq\ \mathit{nm}'\ \rightarrow\ \mathit{field}'\ \mathit{nm}\ (\mathit{update}'\ \mathit{nm}'\ x\ t)\ =\ \mathit{field}'\ \mathit{nm}\ t\\
&\forall\ \mathit{nm} \notin \{(\mathit{ValNm}\ \mathit{Dst}),\ \mathit{nextPCNm}\}.\ \mathit{field}'\ \mathit{nm}\ (\mathit{alu}\ t\ n)\ =\ \mathit{field}'\ \mathit{nm}\ (t\ n)
\end{aligned}$$

The second transaction property can also be stated much more concisely as

$$(s\ =\ t)\ =\ (\forall\ \mathit{nm}.\ \mathit{field}'\ \mathit{nm}\ s\ =\ \mathit{field}'\ \mathit{nm}\ t)$$

Theorems such as these will substantially reduce the amount of work we need to do to prove the desired microarchitecture laws. More importantly, the corresponding Isabelle proof scripts will require significantly fewer changes whenever new transaction fields are added to the transaction ADT. This is because many of the lemmas are implicitly parameterized over a range of field names. Proof steps using those lemmas will automatically cover the new field names. For example, uses of the lemma

$$\forall\ \mathit{nm} \notin \{(\mathit{ValNm}\ \mathit{Dst}),\ \mathit{nextPCNm}\}.\ \mathit{field}'\ \mathit{nm}\ (\mathit{alu}\ t\ n)\ =\ \mathit{field}'\ \mathit{nm}\ (t\ n)$$

will remain valid even after new transaction fields are added to a microarchitecture, provided the *alu* component does not modify the fields.

However, we still use the original typed transaction operators for specifying Hawk circuits, to take advantage of Isabelle's strong type checking. Once we have a well-typed circuit description, we invoke Isabelle's rewriting tactics to automatically transform it into a form that uses field' and update' operations.

The rewriting tactics require a list of already-proven equational theorems that are treated as rewrite rules. We therefore prove such equations for each field accessor and updater. For example, we prove the equational theorem for the dstReg' accessor as follows:

$$\mathit{dstReg}'\ t\ =\ \mathit{castToReg}'\ (\mathit{field}'\ (\mathit{RegNm}\ \mathit{Dst})\ t)$$

$= \{\text{definition of } \mathit{field}' \text{ applied to } (\mathit{RegNm}\ \mathit{Dst})\}$

$\mathit{dstReg}'\ t\ =\ \mathit{castToReg}'\ (\mathit{RegValue}'\ (\mathit{dstReg}'\ t))$

$= \{\text{definition of } \mathit{castToReg}'\}$

$\mathit{dstReg}'\ t\ =\ (\varepsilon\ r.\ \mathit{RegValue}'\ (\mathit{dstReg}'\ t)\ =\ \mathit{RegValue}'\ r)$

$= \{\mathit{RegValue}' \text{ is injective}\}$

$\mathit{dstReg}'\ t\ =\ (\varepsilon\ r.\ \mathit{dstReg}'\ t\ =\ r)$

$= \{\forall y.\ (\varepsilon x.\ y = x) = y\}$

$\mathit{dstReg}'\ t\ =\ \mathit{dstReg}'\ t$

$=$

True

These equational theorems can also be proved automatically using Isabelle's rewriting tactics.

6.3 Lifting the transaction theory to signals

Since Hawk circuits operate on streams of transactions, we find it convenient to define lifted versions of the primitive transaction operators.

$$\begin{aligned}
&\mathit{mkTrans} :: \mathit{Reg} \Rightarrow \mathit{Word} \Rightarrow \mathit{Opcode} \Rightarrow \mathit{Reg} \Rightarrow \mathit{Word} \Rightarrow\\
&\qquad\qquad \mathit{Reg} \Rightarrow \mathit{Word} \Rightarrow \mathit{Word} \Rightarrow \mathit{Word} \Rightarrow \mathit{Trans}\\
&\mathit{mkTrans}\ =\ \mathit{lift}9\ \mathit{mkTrans}'
\end{aligned}$$

$$\begin{aligned}
&\mathit{dstReg}\ ::\ \mathit{Signal}\ \mathit{Trans} \Rightarrow\ \mathit{Signal}\ \mathit{Reg}\\
&\mathit{dstReg}\ =\ \mathit{lift}\ \mathit{dstReg}'
\end{aligned}$$

$$\begin{aligned}
&\mathit{dstValName}\ ::\ \mathit{Signal}\ \mathit{Trans} \Rightarrow\ \mathit{Signal}\ \mathit{Word}\\
&\mathit{dstValName}\ =\ \mathit{lift}\ \mathit{dstValName}'\\
&\vdots
\end{aligned}$$

$$\mathit{setDstReg}\ ::\ \mathit{Signal}\ \mathit{Reg} \Rightarrow\ \mathit{Signal}\ \mathit{Trans} \Rightarrow\ \mathit{Signal}\ \mathit{Reg}$$

$$setDstReg \; = \; lift2\; setDstReg'$$

$$setDstVal \; :: \; Signal\ Word \Rightarrow Signal\ Trans \Rightarrow Signal\ Word$$
$$setDstVal \; = \; lift\ setDstVal'$$
$$\vdots$$

Similarly, the laws governing the transaction operators can also be "lifted" to corresponding laws about the stream-oriented operators. For example, the lifted version of the second transaction property for the $dstReg'$ accessor becomes

$$dstReg\ (mkTrans\ dstReg\ dstValName\ opc\ s1Reg$$
$$s1Val\ s2Reg\ s2Val\ specPC\ brPC) \; = \; dstReg$$

6.4 Proof of *alu* time-invariance for *nop*

We can now define microarchitecture components using the abstract transaction operations. For example, suppose we are defining the *alu* transaction-processing component. Assume that we have already defined the following two functions:

$$arithCore \; :: \; Opcode \Rightarrow Word \Rightarrow Word \Rightarrow Word$$
$$branchCore \; :: \; Opcode \Rightarrow Word \Rightarrow Word \Rightarrow Word \Rightarrow Word$$

Given an opcode describing an arithmetic operation and the values of the two source operands, *arithCore* performs the corresponding arithmetic operation. The *branchCore* function takes an opcode specifying a branch instruction, the values of the two source operands, and the value for the next program counter, and performs the appropriate branch calculation. For instance, if *brIfZero* is the opcode value for the "branch if zero" instruction, then *branchCore brIfZero test addr next* returns *addr* if *test* is equal to zero, otherwise it returns *next*. We leave unspecified what *arithCore* and *branchCore* return when given inappropriate opcodes.

Suppose we also have the two functions *isArithOp* and *isBranchOp*, both of type $Opcode \Rightarrow bool$. The *isArithOp* function returns true when given an arithmetic opcode, and *isBranchOp* returns true when given a branch opcode.

We assume that the *nop* transaction is neither an arithmetic instruction nor a branch instruction:

$$isArithOp\ (opcode'\ nop)\ =\ False$$
$$isBranchOp\ (opcode'\ nop)\ =\ False$$

Given these functions we can define the *alu* component as follows:

```
alu' :: Trans ⇒ Trans
alu' tr =
  let opc = opcode' inp
      s1v = s1Val' inp
      s2v = s2Val' inp
      dstv = if isArithOp opc
                 then arithCore opc s1v s2v
                 else (destVal' inp)
      oldNextPC = nextPC' inp
      nxtPC = if isBranchOp opc
                  then branchCore opc s1v s2v oldNextPC
                  else oldNextPC
  in
      setDstVal' dstv (setNextPC' nxtPC inp)


alu :: Signal Trans ⇒ Signal Trans
alu = lift alu'
```

Now suppose we want to prove that the *alu* circuit is time-invariant for *nop*. That is, we want to prove that for any given transaction signal *inp* that *alu* (*delay nop inp*) is equal to *delay nop* (*alu inp*).

In general, to prove that two signals s and t are equal, we need to prove that for each time n the corresponding signal elements $s\ n$ and $t\ n$ are equal. (where we are considering a signal of type τ to be a function from time to τ.) Thus we must show for each time n that the transaction *alu* (*delay nop inp*) n is equal to the transaction *delay nop* (*alu inp*) n.

We will generalize this statement and prove the following lemma.

Lemma 19 *For all x of type τ, f of type $\tau \Rightarrow \rho$, and xs of type Signal τ, then* *lift f* (*delay x xs*) $=$ *delay* (*f x*) (*lift f xs*)

Proof: We must show for all times n that

$$\mathit{lift}\ f\ (\mathit{delay}\ x\ \mathit{xs})\ n \ = \ \mathit{delay}\ (f\ x)\ (\mathit{lift}\ f\ \mathit{xs})\ n$$

There are two cases to consider: The case where $n = 0$ and the case where $n = k+1$, for some time k:

case $n = 0$**:**

$\mathit{lift}\ f\ (\mathit{delay}\ x\ \mathit{xs})\ 0$

$\quad = \{\text{definition of } \mathit{lift}\}$

$f\ (\mathit{delay}\ x\ \mathit{xs}\ 0)$

$\quad = \{\text{definition of } \mathit{delay}\}$

$f\ x$

$\quad = \{\text{definition of } \mathit{delay}\}$

$\mathit{delay}\ (f\ x)\ (\mathit{lift}\ f\ \mathit{xs})\ 0$

case $n = k+1$**:**

$\mathit{lift}\ f\ (\mathit{delay}\ x\ \mathit{xs})\ (k+1)$

$\quad = \{\text{definition of } \mathit{lift}\}$

$f\ (\mathit{delay}\ x\ \mathit{xs}\ (k+1))$

$\quad = \{\text{definition of } \mathit{delay}\}$

$f\ (\mathit{xs}\ k)$

$\quad = \{\text{definition of } \mathit{lift}\}$

$\mathit{lift}\ f\ \mathit{xs}\ k$

$\quad = \{\text{definition of } \mathit{delay}\}$

$\mathit{delay}\ (f\ x)\ (\mathit{lift}\ f\ \mathit{xs})\ (k+1)$

□

Since $alu = \mathit{lift}\ alu'$, we can use Lemma 19 to prove that alu is time-invariant for nop, provided we show that $alu'\ nop = nop$. To do this we rely on the second property of transactions, and prove that every field of $alu'\ nop$ is equal to the corresponding field of nop. Let $aluModFields$ be the set $\{ValNm\ Dst,\ nextPCNm\}$. From the definition of alu' and the laws for the transaction field accessors and updaters we can derive the following field laws:

$$nm \notin aluModFields \;\rightarrow\; \mathit{field}'\ nm\ (alu'\ t) \;=\; \mathit{field}'\ nm\ t$$

$$\begin{array}{l}
\mathit{field}'\ (ValNm\ Dst)\ (alu'\ t) \;= \\
\quad \mathit{if}\ \ isArithOp\ (castToOpcode'\ (\mathit{field}'\ opcodeNm\ t)) \\
\qquad \mathit{then}\ \ WordValue'\ (arithCore\ (castToOpcode'\ (\mathit{field}'\ opcodeNm\ t)) \\
\qquad\qquad\qquad\qquad\qquad (castToWord'\ (\mathit{field}'\ (ValNm\ Src1)\ t)) \\
\qquad\qquad\qquad\qquad\qquad (castToWord'\ (\mathit{field}'\ (ValNm\ Src2)\ t))) \\
\qquad \mathit{else}\ (\mathit{field}'\ (ValNm\ Dst)\ t)
\end{array}$$

$$\begin{array}{l}
\mathit{field}'\ nextPCNm\ (alu'\ t) \;= \\
\quad \mathit{if}\ \ isBranchOp\ (castToOpcode'\ (\mathit{field}'\ opcodeNm\ t)) \\
\qquad \mathit{then}\ \ WordValue'\ (branchCore\ (castToOpcode'\ (\mathit{field}'\ opcodeNm\ t)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (castToWord'\ (\mathit{field}'\ (ValNm\ Src1)\ t)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (castToWord'\ (\mathit{field}'\ (ValNm\ Src2)\ t)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (castToWord'\ (\mathit{field}'\ nextPC\ t))) \\
\qquad \mathit{else}\ (\mathit{field}'\ nextPCNm\ t)
\end{array}$$

Using these laws and the second property of transactions we can show that alu' preserves every field of nop:

case $nm \notin aluModFields$**:**

$$\mathit{field}'\ nm\ (alu'\ nop) \;=\; \mathit{field}'\ nm\ nop$$

case $nm = (\mathit{ValNm}\ \mathit{Dst})$**:**

$\mathit{field}'\ (\mathit{ValNm}\ \mathit{Dst})\ (\mathit{alu}'\ \mathit{nop})$

$= \{$alu' law for $(\mathit{ValNm}\ \mathit{Dst})\}$

$(\mathit{if}\ \ \mathit{isArithOp}\ (\mathit{castToOpcode}'\ (\mathit{field}'\ \mathit{opcodeNm}\ \mathit{nop}))$

$\quad \mathit{then}\ \mathit{WordValue}'\ (\mathit{arithCore}\ (\mathit{castToOpcode}'\ (\mathit{field}'\ \mathit{opcodeNm}\ \mathit{nop}))$

$\qquad (\mathit{castToWord}'\ (\mathit{field}'\ (\mathit{ValNm}\ \mathit{Src}1)\ \mathit{nop}))$

$\qquad (\mathit{castToWord}'\ (\mathit{field}'\ (\mathit{ValNm}\ \mathit{Src}2)\ \mathit{nop})))$

$\quad \mathit{else}\ (\mathit{field}'\ (\mathit{ValNm}\ \mathit{Dst})\ \mathit{nop}))$

$= \{\mathit{castToOpcode}'\ (\mathit{field}'\ \mathit{opcodeNm}\ t) = \mathit{opcode}'\ t\}$

$(\mathit{if}\ \ \mathit{isArithOp}\ (\mathit{opcode}'\ \mathit{nop})$

$\quad \mathit{then}\ \mathit{WordValue}'\ (\mathit{arithCore}\ (\mathit{castToOpcode}'\ (\mathit{field}'\ \mathit{opcodeNm}\ \mathit{nop}))$

$\qquad (\mathit{castToWord}'\ (\mathit{field}'\ (\mathit{ValNm}\ \mathit{Src}1)\ \mathit{nop}))$

$\qquad (\mathit{castToWord}'\ (\mathit{field}'\ (\mathit{ValNm}\ \mathit{Src}2)\ \mathit{nop})))$

$\quad \mathit{else}\ (\mathit{field}'\ (\mathit{ValNm}\ \mathit{Dst})\ \mathit{nop}))$

$= \{\mathit{isArithOp}\ (\mathit{opcode}'\ \mathit{nop}) = \mathit{False}\}$

$\mathit{field}'\ (\mathit{ValNm}\ \mathit{Dst})\ \mathit{nop}$

case $nm = \mathit{nextPCNm}$**:**

$\mathit{field}'\ \mathit{nextPCNm}\ (\mathit{alu}'\ \mathit{nop})$

$= \{$alu' law for $\mathit{nextPCNm}\}$

$(\mathit{if}\ \ \mathit{isBranchOp}\ (\mathit{castToOpcode}'\ (\mathit{field}'\ \mathit{opcodeNm}\ \mathit{nop}))$

$\quad \mathit{then}\ \mathit{WordValue}'\ (\mathit{branchCore}\ (\mathit{castToOpcode}'\ (\mathit{field}'\ \mathit{opcodeNm}\ \mathit{nop}))$

$\qquad (\mathit{castToWord}'\ (\mathit{field}'\ (\mathit{ValNm}\ \mathit{Src}1)\ \mathit{nop}))$

$\qquad (\mathit{castToWord}'\ (\mathit{field}'\ (\mathit{ValNm}\ \mathit{Src}2)\ \mathit{nop}))$

$\qquad (\mathit{castToWord}'\ (\mathit{field}'\ \mathit{nextPC}\ \mathit{nop})))$

$\quad \mathit{else}\ (\mathit{field}'\ \mathit{nextPCNm}\ \mathit{nop}))$

$= \{\mathit{castToOpcode}'\ (\mathit{field}'\ \mathit{opcodeNm}\ t) = \mathit{opcode}'\ t\}$

$$
\begin{aligned}
&(\mathit{if}\ \ \mathit{isBranchOp}\ (\mathit{opcode}'\ \mathit{nop})\\
&\quad \mathit{then}\ \ \mathit{WordValue}'\ (\mathit{branchCore}\ (\mathit{castToOpcode}'\ (\mathit{field}'\ \mathit{opcodeNm}\ \mathit{nop}))\\
&\qquad\qquad\qquad\qquad (\mathit{castToWord}'\ (\mathit{field}'\ (\mathit{ValNm}\ \mathit{Src}1)\ \mathit{nop}))\\
&\qquad\qquad\qquad\qquad (\mathit{castToWord}'\ (\mathit{field}'\ (\mathit{ValNm}\ \mathit{Src}2)\ \mathit{nop}))\\
&\qquad\qquad\qquad\qquad (\mathit{castToWord}'\ (\mathit{field}'\ \mathit{nextPC}\ \mathit{nop})))\\
&\quad \mathit{else}\ (\mathit{field}'\ \mathit{nextPCNm}\ \mathit{nop}))\\
&\quad = \{\mathit{isBranchOp}\ (\mathit{opcode}'\ \mathit{nop})\ =\ \mathit{False}\}\\
&\mathit{field}'\ \mathit{nextPCNm}\ \mathit{nop}
\end{aligned}
$$

Notice that with the generalized *field′* laws we were able to prove equivalent all of the fields corresponding to the names not in *aluModFields* in one step.

6.5 Temporal reasoning

It is usually necessary to perform induction over time when proving equivalences of components containing internal state, especially when the state-holding elements are part of a feedback loop in the circuit. When performing such proofs, one often has to expose the internal state holding elements and prove properties of them directly. As an example, in the next section we will use inductive reasoning over signals to prove the registerFile-bypass law presented in Section 3.3.3.

6.6 Proving the registerFile-bypass law

We begin by defining the *rf* and *bypass* components in higher order logic, and then state some lemmas about them that will be necessary to the overall proof.

6.6.1 Definition of *envs* and *rf* components

The register file used in the proof follows a write-before-read protocol. On every clock cycle, the contents of the register file are updated by the current value on the writeback input before the file contents are read and sent to the output.

We also designate a special register, called $R0$, as a *zero register*. The contents of $R0$ are hardwired to zero in the instruction set architecture, and writes to $R0$ have no effect. We also stipulate that the register name fields of the *nop* transaction are set to $R0$:

$$\begin{aligned}
&\mathit{regFieldNms} \;=\; \{(\mathit{RegNm}\ \mathit{Dst}),\ (\mathit{RegNm}\ \mathit{Src}1),\ (\mathit{RegNm}\ \mathit{Src}2)\} \\
&\forall\, f \in \mathit{regFieldNms}.\ \mathit{field}'\ f\ \mathit{nop} \;=\; \mathit{RegValue}'\ R0
\end{aligned}$$

The *rf* component is defined in terms of an auxiliary function called *envs*, which is responsible for maintaining the register file contents. The *envs* component outputs the entire contents of the register file on every clock cycle, which the *rf* component then reads when constructing its output transaction. The contents of the register file are represented abstractly as a function of type $\mathit{Reg} \Rightarrow \mathit{Word}$, which we call an *environment*. This representation allows the *envs* component to store the entire register file in a single *delay* component.

We define *envs* below recursively, using the function space over signals CER. The extEnv' helper function modifies an environment by overwriting the contents of a given register, provided it is not $R0$. The *extEnv* function does the same, but is lifted over signals.

We also define the polymorphic *sApply* function, which given a signal of functions and a signal of arguments, applies each function to its corresponding argument and returns the results as a signal. We use *sApply* in the *rf* definition to read the register file contents returned by *envs* on each clock cycle.

$$\mathit{type}\ \mathit{Env}\ = (\mathit{Reg} \Rightarrow\ \mathit{Word})$$

$$\begin{aligned}
&\mathit{extEnv}' \quad :: \ \mathit{Reg} \Rightarrow \mathit{Word} \Rightarrow \mathit{Env} \Rightarrow \mathit{Env} \\
&\mathit{extEnv}' \qquad \mathit{reg}\ \mathit{val}\ \mathit{env}\ = \\
&\quad (\lambda\, r.\ \mathit{if}\ r\ =\ R0\ \mathit{then}\ 0\ \mathit{else}\ \mathit{if}\ \ r\ =\ \mathit{reg} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{then}\ \mathit{val} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ (\mathit{env}\ r))
\end{aligned}$$

$$\mathit{extEnv} \quad :: \ \mathit{Reg}\ \mathit{Signal} \Rightarrow \mathit{Word}\ \mathit{Signal} \Rightarrow \mathit{Env}\ \mathit{Signal} \Rightarrow \mathit{Env}\ \mathit{Signal}$$

$$
\begin{array}{ll}
\mathit{extEnv} & = \mathit{lift3}\ \mathit{extEnv}' \\
\\
\mathit{envs} & ::\ \mathit{Trans\ Signal} \Rightarrow (\mathit{Reg} \Rightarrow \mathit{Word})\ \mathit{Signal} \\
\mathit{envs}\ \mathit{wb} & = \mathit{extEnv}\ (\mathit{dstReg}\ \mathit{wb})\ (\mathit{dstVal}\ \mathit{wb})\ (\mathit{delay}\ (\lambda\, r.\, 0)\ (\mathit{envs}\ \mathit{wb})) \\
\\
\mathit{sApply} & ::\ ('a \Rightarrow\ 'b)\ \mathit{Signal} \Rightarrow\ 'a\ \mathit{Signal} \Rightarrow\ 'b\ \mathit{Signal} \\
\mathit{sApply} & = \mathit{lift2}\ (\lambda\, f\ x.\, f\ x) \\
\\
\mathit{rf} & ::\ \mathit{Trans\ Signal} \Rightarrow \mathit{Trans\ Signal} \Rightarrow \mathit{Trans\ Signal} \\
\mathit{rf} & \quad \mathit{inp}\ \mathit{wb}\ = \\
\end{array}
$$

$$
\begin{array}{l}
\quad \mathit{let}\ \mathit{registers}\ =\ \mathit{envs}\ \mathit{wb} \\
\qquad s1v\ =\ \mathit{sApply}\ \mathit{registers}\ (s1\mathit{Reg}\ \mathit{inp}) \\
\qquad s2v\ =\ \mathit{sApply}\ \mathit{registers}\ (s2\mathit{Reg}\ \mathit{inp}) \\
\quad \mathit{in}\ \ \mathit{setS1Val}\ s1v\ (\mathit{setS2Val}\ s2v\ \mathit{inp})
\end{array}
$$

6.6.2 Converging equivalence relations for signals

Like many stateful components in Hawk, the *envs* component is defined as a recursive equation over signals. To ensure that this definition is consistent we need to demonstrate that the equation has a unique solution. We do this by defining a converging equivalence relation (CER) for signals, and then show that the fixpoint functional associated with *envs*'s definition is contracting.

Recalling Chapter 5, a CER contains four components: a resolution type ρ, a target type τ, a well-founded, transitive relation $(<)$ of type $\rho \Rightarrow \rho \Rightarrow \mathit{bool}$ called the resolution ordering, and an indexed equivalence relation $(\approx)$ of type $\rho \Rightarrow \tau \Rightarrow \tau \Rightarrow \mathit{bool}$. We can define a signal CER that is similar to the lazy list CER, with *nat* for the resolution type, and the usual less-than ordering on the naturals for $(<)$. The target type is the type of signals, which we are modeling as functions indexed on the naturals, $(\mathit{nat} \Rightarrow\ 'a)$. The indexed equivalence relation is defined as:

$$
f \overset{n}{\approx} g \quad \equiv \quad (\forall\, i.\ i < n \rightarrow f\ i = g\ i)
$$

In other words, two signals f and g are equivalent at resolution n if their first $n-1$ elements are equal.

The first six CER axioms are easy to verify with these definitions. The last two axioms can be proved with the following existential witness elements, respectively:

$$\begin{aligned} \mathit{local\_signal\_limit}\ F\ i &\equiv F\ (i-1) \\ \mathit{global\_signal\_limit}\ F &\equiv (\lambda\, n.\ F\ (n+1)\ n) \end{aligned}$$

Proofs of the last two CER axioms involve, at some point, choosing an arbitrary pair of resolutions i and j such that $j < i$, and then performing a case analysis on whether $j = i - 1$.

Equivalences for the *lift* primitives

The family of lift primitives *lift*, *lift*2, ..., and the *delay* primitive can be abstractly characterized as conditional *equivalence* laws that specify how they preserve the $(\approx)$ relation. These equivalences can be used to prove that cyclic circuits like *envs* are contracting, without having to expand the definitions of the primitives.

The *lift* primitive is a combinational circuit, so its output value at any time n is dependent on its input value at that same time value:

$$xs \overset{n}{\approx} ys \quad \rightarrow \quad \mathit{lift}\ f\ xs \ \overset{n}{\approx}\ \mathit{lift}\ f\ ys \tag{6.1}$$

Proof: Assume the antecedent $xs \overset{n}{\approx} ys$. Expanding the definition of $(\approx)$, this is equivalent to assuming $(\forall\, i.\ i < n \rightarrow xs\ i = ys\ i)$. Expanding the definition of $(\approx)$ on the consequent side of the formula, we must show for arbitrary $i < n$ that $\mathit{lift}\ f\ xs\ i = \mathit{lift}\ f\ ys\ i$. By definition of *lift* this goal is equivalent to showing $f\ (xs\ i) = f\ (ys\ i)$. But this is true since $xs\ i = ys\ i$ by assumption $\Box$

By similar reasoning every lift_k primitive can be characterized as

$$xs_1 \overset{n}{\approx} ys_1 \ \wedge\ \ldots\ \wedge\ xs_k \overset{n}{\approx} ys_k \quad \rightarrow \quad \mathit{lift}_k\ f\ xs_1\ \ldots\ xs_k\ \overset{n}{\approx}\ \mathit{lift}_k\ f\ ys_1\ \ldots\ ys_k$$

The body of *envs* makes use of the auxiliary functions *extEnv*, *dstReg*, and *dstValName*.

All of these auxiliaries are defined in terms of $lift_k$ primitives, and therefore obey the following signal CER equivalences:

$$\begin{aligned} ws \overset{n}{\approx} xs \ \wedge\ ys \overset{n}{\approx} zs &\rightarrow\ extEnv\ ws\ ys\ \overset{n}{\approx}\ extEnv\ xs\ zs \\ xs \overset{n}{\approx} ys &\rightarrow\ dstReg\ xs\ \overset{n}{\approx}\ dstReg\ ys \\ xs \overset{n}{\approx} ys &\rightarrow\ dstVal\ xs\ \overset{n}{\approx}\ dstVal\ ys \end{aligned}$$

Equivalence for the *delay* primitive

The *delay* component is a contracting function for the signal CER, which accords with the intuition that every feedback cycle in a well formed circuit definition must contain at least one delay:

$$xs \overset{n}{\approx} ys \ \ \rightarrow\ \ delay\ z\ xs \overset{n+1}{\approx} delay\ z\ ys$$

Notice that the initial value parameters to both delay components have to be equal for the equivalence to hold.

Proof: Assume the antecedent $xs \overset{n}{\approx} ys$. This is equivalent to assuming $(\forall\ i.\ i < n \rightarrow xs\ i = ys\ i)$. Expanding $(\approx)$ in the consequent, we must show for arbitrary $i < n+1$ that $delay\ z\ xs\ i = delay\ z\ ys\ i$. If $i = 0$, this reduces by the definition of *delay* to showing that z = z, which is true. If $i > 0$, then the consequent reduces to showing that $xs\ (i-1) = ys\ (i-1)$, which is true by assumption since $i - 1 < n$ $\Box$

Proving *envs* is contracting

Now that we have equivalences for all of the functions in the body of *envs*, we need to show that the recursive definition of *envs* itself is consistent by showing that it is contracting over some CER. Since the definition is parameterized on the argument *wb*, we show that the fixpoint functional

$$F \ \equiv\ (\lambda\ envs'\ wb.\ extEnv\ (dstReg\ wb)\ (dstVal\ wb)\ (delay\ (\lambda\ r.\ 0)\ (envs'\ wb)))$$

derived from the recursion equation for *envs* is contracting on the function space over signals CER defined in Section 5.5. This lifted CER still uses the $(<_{nat})$ of the signal

CER, so by (5.10) of Section 5.4 and the definition of $(\approx)$ for the function space CER combinator it suffices to show for arbitrary resolution i and functions g and h that

$$(\forall\ xs.\ g\ xs \overset{i}{\approx} h\ xs) \quad \rightarrow \quad \forall\ xs.\ F\ g\ xs \overset{i+1}{\approx} F\ h\ xs$$

We assume the antecedent $(\forall\ xs.\ g\ xs \overset{i}{\approx} h\ xs)$ and prove the consequent for arbitrary xs by applying the appropriate equivalences:

$F\ g\ xs \overset{i+1}{\approx} F\ h\ xs$

$\quad = \{\text{Definition of } F\}$

$\mathit{extEnv}\ (\mathit{dstReg}\ xs)\ (\mathit{dstValName}\ xs)\ (\mathit{delay}\ (\lambda\ r.\ 0)\ (f\ xs)) \overset{i+1}{\approx}$
$\mathit{extEnv}\ (\mathit{dstReg}\ xs)\ (\mathit{dstValName}\ xs)\ (\mathit{delay}\ (\lambda\ r.\ 0)\ (g\ xs))$

$\quad \Leftarrow \{\text{Equivalence law for } \mathit{extEnv}\}$

$(\mathit{dstReg}\ xs) \overset{i+1}{\approx} (\mathit{dstReg}\ xs)\ \wedge$
$(\mathit{dstValName}\ xs) \overset{i+1}{\approx} (\mathit{dstValName}\ xs)\ \wedge$
$(\mathit{delay}\ (\lambda\ r.\ 0)\ (f\ xs)) \overset{i+1}{\approx} (\mathit{delay}\ (\lambda\ r.\ 0)\ (g\ xs))$

$\quad = \{(\approx) \text{ is reflexive at all resolutions by CER axiom (5.1)}\}$

$(\mathit{delay}\ (\lambda\ r.\ 0)\ (f\ xs)) \overset{i+1}{\approx} (\mathit{delay}\ (\lambda\ r.\ 0)\ (g\ xs))$
$\quad \Leftarrow \{\text{Equivalence law for } \mathit{delay} \text{ component}\}$
$(f\ xs) \overset{i}{\approx} (g\ xs)$
$\quad = \{\text{Assumption}\}$
True

Demonstrating that recursive Hawk circuits like *envs* are contracting can usually be proved within Isabelle in a couple of steps, by relying on Isabelle's high-level rewriting and tableau decision procedures. The result of invoking the decision procedures is the recursive equation for *envs* proved as a certified theorem.

6.6.3 Properties of *envs* component

The core of the register file-bypass verification involves proving various properties of the recursive *envs* function. There are five basic properties of *envs* needed in the top-level proof. The first two *envs* properties state that $R0$ is a zero register.

$$envs\ wb\ n\ R0\ =\ 0$$

$$dstReg'\ (wb\ (n+1))\ =\ R0\ \rightarrow\ envs\ wb\ (n+1)\ r\ =\ envs\ wb\ n\ r$$

The third *envs* property states that the environment returned by *envs* on a given clock cycle has been updated correctly with respect to the current *wb* transaction.

$$dstReg'\ (wb\ n)\ \neq\ R0\ \rightarrow\ envs\ wb\ n\ (dstReg'\ (wb\ n))\ =\ dstValName'\ (wb\ n)$$

The fourth and fifth *envs* properties deal with register values that are not being written to on the current cycle. The fourth property states that initially every register not currently being written to is zeroed out.

$$r\ \neq\ dstReg'\ (wb\ 0)\ \rightarrow\ envs\ wb\ 0\ r\ =\ 0$$

The fifth *envs* property states that at every cycle after the initial cycle, every register that is not currently being written to is equal to the value it had on the previous cycle.

$$r\ \neq\ dstReg'\ (wb\ (n+1))\ \rightarrow\ envs\ wb\ (n+1)\ r\ =\ envs\ wb\ n\ r$$

The above properties can be proved by unwinding the definitions of *envs* and its constituents. For example, we prove the last property as follows:

Assume $r\ \neq\ dstReg'\ (wb\ (n+1))$. **Then**

case $r\ =\ R0$**:**

$envs\ wb\ (n+1)\ R0$

$\quad = \{\text{First property of } envs\}$

0

$\quad = \{\text{First property of } envs\}$

$$envs\ wb\ n\ R0$$

case $r \neq R0$**:**

$$envs\ wb\ (n+1)\ r$$

$$= \{\text{Definition of } envs\}$$

$$\begin{aligned}&extEnv\ (dstReg\ wb)\\ &\qquad (dstValName\ wb)\\ &\qquad (delay\ (\lambda\, r.\, 0)\ (envs\ wb))\ (n+1)\ r\end{aligned}$$

$$= \{\text{Definitions of } extEnv, lift4, dstReg, dstVal, lift, delay\}$$

$$\begin{aligned}&extEnv'\ (dstReg'\ (wb\ (n+1)))\\ &\qquad (dstValName'\ (wb\ (n+1)))\\ &\qquad (envs\ wb\ n)\ r\end{aligned}$$

$$= \{\text{Definition of } extEnv'\}$$

$$\begin{aligned}(&if\ r\ =\ R0\ then\ 0\ else\ if\ r\ =\ (dstReg'\ (wb\ (n+1)))\\ &\qquad then\ (dstValName'\ (wb\ (n+1)))\\ &\qquad else\ (envs\ wb\ n\ r))\end{aligned}$$

$$= \{r \neq R0 \wedge r \neq dstReg'(wb(n+1))\}$$

$$envs\ wb\ n\ r$$

The last two properties of *envs* can be stated as a single theorem by using the *delay* component.

$$r\ \neq\ dstReg'\ (wb\ n)\ \rightarrow\ envs\ wb\ n\ r\ =\ envs\ (delay\ nop\ wb)\ n\ r$$

The proof proceeds by induction on time values n, and a compound case analysis on the values of r and wb:

Assume $r\ \neq\ dstReg'\ (wb\ n)$**. Then**

case $r\ =\ R0$**:**

$$envs\ wb\ n\ R0$$

$$= \{\text{First } envs \text{ property}\}$$

$$0$$

$= \{\text{First } \mathit{envs} \text{ property}\}$

$\mathit{envs}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb})\ n\ R0$

case $r \neq R0 \wedge n = 0$**:**

$\mathit{envs}\ \mathit{wb}\ 0\ r$

$= \{\text{Assumption and fourth } \mathit{envs} \text{ property}\}$

0

$= \{r \neq \mathit{dstReg}'\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb}\ 0);\ \text{fourth } \mathit{envs} \text{ property}\}$

$\mathit{envs}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb})\ 0\ r$

case $r \neq R0 \wedge n = (k+1) \wedge r = \mathit{dstReg}'\ (\mathit{wb}\ k)$, for some k**:**

$\mathit{envs}\ \mathit{wb}\ (k+1)\ r$

$= \{\text{Assumption and fifth } \mathit{envs} \text{ property}\}$

$\mathit{envs}\ \mathit{wb}\ k\ r$

$= \{\text{Third } \mathit{envs} \text{ property}\}$

$\mathit{dstValName}'\ (\mathit{wb}\ k)$

$= \{\text{Definition of } \mathit{delay}\}$

$\mathit{dstValName}'\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb}\ (k+1))$

$= \{r = \mathit{dstReg}'\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb}\ (k+1));\ \text{Third } \mathit{envs} \text{ property}\}$

$\mathit{envs}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb})\ (k+1)\ r$

case $r \neq R0 \wedge n = (k+1) \wedge r \neq \mathit{dstReg}'\ (\mathit{wb}\ k)$, for some k**:**

Inductive hypothesis:

$r \neq \mathit{dstReg}'\ (\mathit{wb}\ k) \rightarrow \mathit{envs}\ \mathit{wb}\ k\ r = \mathit{envs}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb})\ k\ r$.

Then

$\mathit{envs}\ \mathit{wb}\ (k+1)\ r$

$= \{\text{Assumption and fifth } \mathit{envs} \text{ property}\}$

$\mathit{envs}\ \mathit{wb}\ k\ r$

$= \{\text{ind. hyp.}\}$

$\mathit{envs}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb})\ k\ r$

$= \{r \neq \mathit{dstReg}'\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb}\ (k+1))$; fifth *envs* property$\}$

$\mathit{envs}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb})\ (k+1)\ r$

6.6.4 Definition and properties of *fvEnvs* component

The registerFile-bypass proof makes heavy use of the *field'* function, which operates over *FieldValues*. We can simplify the proof somewhat by introducing an alternate version of the *envs* function, called *fvEnvs*, that returns environments of type $\mathit{FieldValue} \Rightarrow \mathit{FieldValue}$. The use of *fvEnvs* removes the need to insert cast operations when applying an environment to the *RegValue'* returned by a *field'* operation.

$$\mathit{type}\ \mathit{FvEnv} = \mathit{FieldValue} \Rightarrow \mathit{FieldValue}$$

$$\begin{aligned}
&\mathit{fvEnvs}' && :: \mathit{Env} \Rightarrow \mathit{FvEnv} \\
&\mathit{fvEnvs}'\ \mathit{env} && = (\lambda\, \mathit{fv}.\ \mathit{WordValue}'\ (\mathit{env}\ (\mathit{castToReg}'\ \mathit{fv})))
\end{aligned}$$

$$\begin{aligned}
&\mathit{fvEnvs} && :: \mathit{Trans}\ \mathit{Signal} \Rightarrow \mathit{FvEnv}\ \mathit{Signal} \\
&\mathit{fvEnvs}\ \mathit{wb} && = \mathit{lift}\ \mathit{fvEnvs}'\ (\mathit{envs}\ \mathit{wb})
\end{aligned}$$

The properties proved of *envs* carry over to *fvEnvs*. For example, the delay law for *fvEnvs* becomes

$$\begin{aligned}
&(\mathit{RegValue}'\ r)\ \neq\ \mathit{field}'\ (\mathit{RegNm}\ \mathit{Dst})\ (\mathit{wb}\ n)\ \rightarrow \\
&\quad \mathit{fvEnvs}\ \mathit{wb}\ n\ (\mathit{RegValue}'\ r)\ =\ \mathit{fvEnvs}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb})\ n\ (\mathit{RegValue}'\ r).
\end{aligned}$$

6.6.5 Definition and properties of *bypass* component

All that remains before we tackle the main registerFile-bypass proof is to define *bypass* and derive its characteristic properties. The *bypass* component is defined in terms of the auxiliary function *bypassSelect*, which performs the bypass operation on a single operand value.

$$\mathit{bypassSelect}'\ ::\ \mathit{Reg} \Rightarrow \mathit{Word} \Rightarrow \mathit{Reg} \Rightarrow \mathit{Word} \Rightarrow \mathit{Word}$$

$$
\begin{array}{l}
\textit{bypassSelect}'\ \textit{inpReg}\ \textit{inpWord}\ \textit{wbReg}\ \textit{wbWord}\ =\\
\quad \textit{if}\ \ \textit{inpReg}\ =\ R0\ \vee\ \textit{wbReg} \neq \textit{inpReg}\\
\quad\quad \textit{then}\ \textit{inpWord}\\
\quad\quad \textit{else}\ \textit{wbWord}
\end{array}
$$

$$
\begin{array}{l}
\textit{bypassSelect}\ :: \textit{Reg Signal} \Rightarrow \textit{Word Signal} \Rightarrow \textit{Reg Signal} \Rightarrow \textit{Word Signal} \Rightarrow\\
\qquad\qquad\qquad \textit{Word Signal}\\
\textit{bypassSelect}\ =\ \textit{lift4}\ \textit{bypassSelect}'
\end{array}
$$

$$
\begin{array}{l}
\textit{bypass}\ ::\ \textit{Trans Signal} \Rightarrow \textit{Trans Signal} \Rightarrow \textit{Trans Signal}\\
\textit{bypass}\ \textit{input}\ \textit{writeback}\ =\\
\quad \textit{let}\ \textit{wbReg}\ =\ \textit{dstReg}\ \textit{writeback}\\
\quad\quad \textit{wbVal}\ =\ \textit{dstValName}\ \textit{writeback}\\
\quad\quad s1v\ =\ \textit{bypassSelect}\ (s1\textit{Reg}\ \textit{input})\ (s1\textit{Val}\ \textit{input})\ \textit{wbReg}\ \textit{wbVal}\\
\quad\quad s2v\ =\ \textit{bypassSelect}\ (s2\textit{Reg}\ \textit{input})\ (s2\textit{Val}\ \textit{input})\ \textit{wbReg}\ \textit{wbVal}\\
\quad \textit{in}\ \ \textit{setS1Val}\ s1v\ (\textit{setS2Val}\ s2v\ \textit{input})
\end{array}
$$

The properties we derive from the definition of *bypass* are that the component does not modify any input transaction field other than the two source operand values:

$$
\begin{array}{l}
\forall\ f \notin \{\textit{ValNm}\ \textit{Src}1,\ \textit{ValNm}\ \textit{Src}2\}.\\
\quad \textit{field}'\ f\ (\textit{bypass}\ \textit{inp}\ \textit{wb}\ n)\ =\ \textit{field}'\ f\ (\textit{inp}\ n)
\end{array}
$$

and that *bypass* performs correctly on the source operand values:

$$
\forall\ i \in \{\textit{Src}1,\ \textit{Src}2\}.
$$

$$
\begin{array}{l}
let\ \ inpReg\ =\ \mathit{field}'\ (RegNm\ i)\ (inp\ n) \\
\quad\ \ wbReg\ =\ \mathit{field}'\ (RegNm\ Dst)\ (wb\ n) \\
in\ ((\ inpReg\ =\ (RegValue'\ R0)\ \vee\ inpReg \neq wbReg\ \rightarrow \\
\qquad \mathit{field}'\ (ValNm\ i)\ (bypass\ inp\ wb\ n)\ =\ \mathit{field}'\ (ValNm\ i)\ (inp\ n)) \\
\quad\ \ \wedge \\
\quad\ \ (\ inpReg \neq (RegValue'\ R0)\ \wedge\ inpReg\ =\ wbReg\ \rightarrow \\
\qquad \mathit{field}'\ (ValNm\ i)\ (bypass\ inp\ wb\ n)\ =\ \mathit{field}'\ (ValNm\ Dst)\ (wb\ n)))
\end{array}
$$

These properties can be proved by expanding the definitions of $bypass$, $bypassSelect$, and $bypassSelect'$, and then performing a case analysis on the appropriate register name fields of inp and wb.

6.6.6 Proof of the microarchitecture law

Now that we have the needed $fvEnvs$ and $bypass$ properties, the top level proof itself is relatively straightforward. The formal statement of the theorem is as follows:

$$bypass\ (rf\ inp\ (delay\ nop\ wb))\ wb\ =\ rf\ inp\ wb$$

We prove these two signals equal by showing that they are equal at all time periods n, for all transaction fields $f \in FieldNm$:

case $f \in FieldNm - \{(ValNm\ Src1),\ (ValNm\ Src2)\}$**:**

$$
\begin{array}{l}
\mathit{field}'\ f\ (bypass\ (rf\ inp\ (delay\ nop\ wb))\ wb\ n) \\
\quad = \{bypass \text{ doesn't modify field } f\} \\
\mathit{field}'\ f\ (rf\ inp\ (delay\ nop\ wb)\ n) \\
\quad = \{rf \text{ doesn't modify field } f\} \\
\mathit{field}'\ f\ (inp\ n) \\
\quad = \{rf \text{ doesn't modify field } f\} \\
\mathit{field}'\ f\ (rf\ inp\ wb\ n)
\end{array}
$$

case $f = \mathit{ValNm}\ i$, for $i \in \{\mathit{Src}1,\ \mathit{Src}2\}$**:**

subcase $\mathit{field}'\ (\mathit{RegNm}\ i)\ (\mathit{inp}\ n) = \mathit{RegValue}'\ R0$**:**

$\mathit{field}'\ (\mathit{ValNm}\ i)\ (\mathit{bypass}\ (\mathit{rf}\ \mathit{inp}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb}))\ \mathit{wb}\ n)$

$=$ {Subcase assumption; *bypass* preserves zero registers of input}

$\mathit{field}'\ (\mathit{ValNm}\ i)\ (\mathit{rf}\ \mathit{inp}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb})\ n)$

$=$ {Subcase assumption; *rf* preserves zero registers of transactions}

$\mathit{WordValue}'\ 0$

$=$ {Subcase assumption; *rf* preserves zero registers of transactions}

$\mathit{field}'\ (\mathit{ValNm}\ i)\ (\mathit{rf}\ \mathit{inp}\ \mathit{wb}\ n)$

subcase $\mathit{field}'\ (\mathit{RegNm}\ i)\ (\mathit{inp}\ n) \neq \mathit{RegValue}'\ R0\ \wedge$

$\mathit{field}'\ (\mathit{RegNm}\ i)\ (\mathit{inp}\ n) = \mathit{field}'\ (\mathit{RegNm}\ \mathit{Dst})\ (\mathit{wb}\ n)$**:**

L.H.S. :

$\mathit{field}'\ (\mathit{ValNm}\ i)\ (\mathit{bypass}\ (\mathit{rf}\ \mathit{inp}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb}))\ \mathit{wb}\ n)$

$=$ {*rf* preserves *RegNm* i field; *bypass* overwrite law}

$\mathit{field}'\ (\mathit{ValNm}\ \mathit{Dst})\ (\mathit{wb}\ n)$.

R.H.S. :

$\mathit{field}'\ (\mathit{ValNm}\ i)\ (\mathit{rf}\ \mathit{inp}\ \mathit{wb}\ n)$

$=$ {Definition of *rf*; *field* and *update* laws}

$\mathit{sApply}\ (\mathit{fvEnvs}\ \mathit{wb})\ (\mathit{field}\ (\mathit{RegNm}\ i)\ \mathit{inp})\ n$

$=$ {Lift laws}

$\mathit{fvEnvs}'\ \mathit{wb}\ n\ (\mathit{field}'\ (\mathit{RegNm}\ i)\ (\mathit{inp}\ n))$

$=$ {Subcase assumptions; third *envs* property}

$\mathit{field}'\ (\mathit{ValNm}\ \mathit{Dst})\ (\mathit{wb}\ n)$.

subcase $\mathit{field}'\ (\mathit{RegNm}\ i)\ (\mathit{inp}\ n) \neq \mathit{RegValue}'\ R0\ \wedge$

$\mathit{field}'\ (\mathit{RegNm}\ i)\ (\mathit{inp}\ n) \neq \mathit{field}'\ (\mathit{RegNm}\ \mathit{Dst})\ (\mathit{wb}\ n)$**:**

$\mathit{field}'\ (\mathit{ValNm}\ i)\ (\mathit{bypass}\ (\mathit{rf}\ \mathit{inp}\ (\mathit{delay}\ \mathit{nop}\ \mathit{wb}))\ \mathit{wb}\ n)$

$=$ {*rf* preserves *RegNm* i field; *bypass* no-overwrite law}

$$\textit{field}'\ (\textit{ValNm}\ i)\ (\textit{rf}\ \textit{inp}\ (\textit{delay}\ \textit{nop}\ \textit{wb})\ n).$$
$$= \{\text{Subcase assumptions; } \textit{rf} \text{ delay law}\}$$
$$\textit{field}'\ (\textit{ValNm}\ i)\ (\textit{rf}\ \textit{inp}\ \textit{wb}\ n)$$

Thus, with some work we've been able to algebraically verify the important register file - bypass law. The other microarchitecture law proofs, especially those involving circuits with cyclic state holding elements, use similar techniques. That is, the original circuits are generalized to circuits where all internal state elements are visible. The generalized circuits are proved equivalent by induction over time. The microarchitecture law then holds as a special case. Section 7.5 of the next chapter discusses our efforts to mechanize the microarchitecture laws and pipeline simplifications.

Chapter 7

Retrospective

On page one of the 1988 textbook *Introduction to Functional Programming*, Bird and Wadler[7] summarize one of the primary motivations behind using a pure functional language as a means for creating executable specifications:

> A characteristic feature of functional programming is that if an expression possesses a well-defined value, then the order in which a computer may carry out the evaluation does not affect the outcome. In other words, the meaning of an expression is its value and the task of the computer is simply to obtain it. It follows that expressions in a functional language can be constructed, manipulated and reasoned about, like any other kind of mathematical expression, using more or less familiar algebraic laws. The result, as we hope to justify, is a conceptual framework for programming which is at once very simple, very concise, very flexible and very powerful.

One can view this thesis as a case study for Bird and Wadler's programme, demonstrating that functional specification languages and algebraic reasoning can feasibly model domains of a useful size, in this case pipelined processor microarchitectures. The rest of this chapter evaluates the merits of this approach. In particular, we will examine the strengths and weaknesses of

- Using a functional programming language as the basis of a high-level hardware description language.
- Transactions as a microarchitectural structuring principle.

- The algebraic approach to pipeline transformation and verification, and its mechanization in Isabelle.

We will also discuss the usefulness of converging equivalence relations as a general mechanism for defining recursive values in higher order logic.

7.1 The functional basis of Hawk

This section discusses the benefits and limitations of Hawk's functional basis as we encountered them during the course of this thesis. Although the decision to make Hawk an embedded language within Haskell imposed some restrictions, in general Haskell's collection of functional language features allowed us to specify microarchitectures at an impressively high level of abstraction.

7.1.1 Structured datatypes

Algebraic datatypes and pattern matching were used extensively when specifying the *alu* and *mem* components of the DLX microarchitecture. The Haskell functions implementing these components have to perform a series of tests on the opcode field to determine what exact operation to perform. Even though the DLX architecture is built around a simplified RISC instruction set, the meaning of an opcode can still become quite involved. The Hawk team used a hierarchical collection of algebraic datatypes to represent opcode values, and used nested pattern matching to perform the necessary tests.

$$
\begin{aligned}
\mathit{data\ Opcode}\ =\ & \mathit{ExecOp\ AluOp} \\
\mid\ & \mathit{MemOp\ LoadStoreOp} \\
\mid\ & \ldots
\end{aligned}
$$

$$
\begin{aligned}
\mathit{data}\ \mathit{AluOp}\ =\ & \mathit{Add}\ \mathit{Signedness} \\
& \mid \mathit{Sub}\ \mathit{Signedness} \\
& \mid \mathit{Mult}\ \mathit{Signedness} \\
& \mid \mathit{Div}\ \mathit{Signedness} \\
& \mid \mathit{And} \\
& \mid \mathit{Or} \mid \mathit{Xor} \\
& \mid \mathit{ShiftLL} \mid \mathit{ShiftRL} \mid \mathit{ShiftRA} \\
& \mid \mathit{Cmp}\ \mathit{Comparison} \\
& \mid \ldots
\end{aligned}
$$

$$
\mathit{data}\ \mathit{Signedness}\ =\ \mathit{Signed} \mid \mathit{Unsigned}
$$

$$
\begin{aligned}
\mathit{data}\ \mathit{Comparison}\ =\ & \mathit{LessThan} \\
& \mid \mathit{LessEqual} \\
& \mid \mathit{GreaterThan} \\
& \mid \mathit{GreaterEqual} \\
& \mid \mathit{Equal} \\
& \mid \mathit{NotEqual}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{data}\ \mathit{LoadStoreOp}\ =\ & \mathit{Load}\ \mathit{WordSize}\ \mathit{Signedness} \\
& \mid \mathit{Store}\ \mathit{WordSize} \\
& \mid \mathit{NOP}
\end{aligned}
$$

$$
\mathit{data}\ \mathit{WordSize}\ =\ \mathit{Byte} \mid \mathit{HalfWord} \mid \mathit{FullWord}
$$

In lower-level hardware description languages, these opcode values would simply be laid out as a single bit-vector, or perhaps as an unstructured collection of scalar variables. In this case, the designer of the *alu* and *mem* decoding logic would have to be careful to select the correct bitfield subranges or scalar variables using nested conditionals. Even when using scalar variables it is often the case that the meaning of some variables depends on the values contained in other variables. For example, a “wordsize” variable would have no meaning if the arithmetic opcode variable is set to “Xor”, since in the DLX the exclusive-or operation is always performed at full word size.

It is quite easy to make a mistake in such situations, even for instruction sets as simple as the DLX. While Hawk's type system will catch incorrect pattern-match expressions automatically, lower level hardware languages typically do not enforce subrange boundaries, nor do they provide any way to state that the interpretation of one variable is dependent on the value held in another. In these languages such mistakes have to be debugged at runtime. The situation becomes even worse when several designers are responsible for decoding portions of the instruction.

7.1.2 Lazy evaluation

Formally we model signals as functions over time, but in a simulation implementation Hawk signals are implemented as lazy infinite lists. This design choice is essential if we want to implement shared signal values efficiently. Consider the following Hawk circuit

$$\begin{aligned} \mathit{fib} &= \mathit{delay}\;1\;(\mathit{lift}2\;(+)\;\mathit{fib}\;\mathit{fib}') \\ \mathit{fib}' &= \mathit{delay}\;1\;\mathit{fib} \end{aligned}$$

which calculates the Fibonacci sequence $[1, 1, 2, 3, 5, \ldots]$. Notice that the *fib* signal is referenced twice in a feedback loop: once as an argument to $\mathit{lift}2$, and once as an argument to *delay* in the definition of fib'. If we use lazy lists, Haskell's lazy evaluation strategy will calculate a given element of this sequence once, if needed to evaluate a client expression, and then store the result in memory, so that subsequent references to the element are evaluated in constant time. What this means for the *fib* sequence is that it takes at most $\mathbf{O}(n)$ accesses to compute the n^{th} element of the signal, since lower-numbered *fib* elements are effectively cached in the runtime heap.

On the other hand, if we had implemented signals as functions the optimized code eventually generated by Haskell for *fib* would have looked something like this:

$$\begin{aligned} \mathit{fib}\;0 &= 1 \\ \mathit{fib}\;1 &= 1 \\ \mathit{fib}\;n &= \mathit{fib}\;(n-1) \;+\; \mathit{fib}\;(n-2) \end{aligned}$$

The two recursive calls to *fib* mean that every call to *fib* n takes $\mathbf{O}(2^n)$ recursive calls to evaluate. This exponential blowup in evaluation time happens whenever a shared signal is referenced in two or more places within a feedback loop, as *fib* is. In essence, Haskell's lazy evaluation mechanism applied to lists automatically implements a form of dynamic programming.

7.1.3 Higher order functions

Haskell's ability to manipulate functions as first class values not only allows us to conveniently map functions over signals through the *lift* primitives, it also allows common wiring patterns to be encapsulated as higher order Hawk components. For example, Cook et al[18] describe a parameterized reservation station component as part of a superscalar out-of-order microarchitecture. The reservation station component *station* takes a signal of unordered collections of transactions and sends each transaction to an appropriate execution unit, if one is available. If no execution unit is available, the reservation station stores the transaction in an internal *reservation buffer* until an execution unit becomes free.

To increase its generality the *station* component is parameterized on a list of execution units *execUnits*, among other things. Each execution unit is a function that takes a reset signal and a signal of transaction collections (each transaction collection is implemented as a list of transactions). The execution unit returns two signals of transaction collections: The first signal consists of transactions that the execution unit refused to process, either because it is already processing a transaction or because the transaction is of the wrong type. The second signal contains transactions that the execution unit has completed on that clock cycle. The Hawk code for the reservation station component is sketched below:

$$type\ ExecUnit\ =\ Signal\ Bool \rightarrow Signal\ [Trans] \rightarrow (Signal\ [Trans], Signal\ [Trans])$$

$$station\ ::\ (Int,\ [ExecUnit]) \rightarrow (Signal\ Bool,\ Signal\ [Trans]) \rightarrow Signal\ [Trans]$$
$$station\ (numReservations, execUnits)\ (reset,\ inputTransactions)\ =\ \ldots$$

Since the execution units are themselves functions, the *station* component is an example of a higher order component. In practice it is quite useful to be able to vary the type and number of execution units given to *station*, without having to change the reservation station's definition.

The top-level pipelined microarchitecture considered in this thesis did not have a regular enough structure to significantly benefit from exploiting the higher order features of Hawk, other than using the *lift* primitives. We did use higher order functions to formally model the contents of the register file component in Section 6.6.1.

7.1.4 Static typing and polymorphism

The Hawk team has relied extensively on Haskell's static typing enforcement to quickly catch coding mistakes when implementing microarchitectures. Without explicit type checking, errors that normally took us seconds to find and fix could have taken minutes or hours to debug at runtime. It is particularly easy in modeling to forget the difference between a static value of type τ and a dynamic signal whose elements are of type τ. For example, the following Hawk code is ill-typed, and is quickly rejected by the type checker:

$$
\begin{aligned}
&\mathit{select} \; :: \; \mathit{Signal\ Bool} \rightarrow \mathit{Signal}\ \alpha \rightarrow \mathit{Signal}\ \alpha \rightarrow \mathit{Signal}\ \alpha \\
&\mathit{resultReg} \; :: \; \mathit{Signal\ Reg} \\
&\mathit{resultValid} \; :: \; \mathit{Signal\ Bool} \\
\\
&\mathit{finalResult} \; :: \; \mathit{Signal\ Reg} \\
&\mathit{finalResult} \; = \; \mathit{select\ resultValid\ resultReg}\ R0
\end{aligned}
$$

The type error that Hugs 98 (a Haskell interpreter) prints out is:

```
ERROR (line 20): Type error in application
*** Expression     : select resultValid resultReg R0
*** Term           : R0
*** Type           : Reg
*** Does not match : Signal a
```

The type checker is pointing out that in the definition of *finalResult*, $R0$ is a static register name, not a signal as required by *select*. If Hawk was a dynamically typed language, this error would not have been detected until *select* was evaluated at a clock cycle where *resultValid* was false. If this is a rare occurrence, then it could be quite a while before the bug is even detected.

Explicit type annotations have also been quite helpful as a form of machine-checked documentation. Several of the microarchitectures in the Hawk library have been initially designed by one person and then enhanced or maintained by another. We invariably find that explicitly typed code is easier to understand by other team members.

While Haskell's polymorphic type system admits a wide range of useful programs while automatically inferring general types for them, there are conceptually valid microarchitecture designs that Haskell can only type check by adding explicit typecasts[1]. For example, a module may be implementing a shared bus with a signal of heterogeneously-typed elements. At any given clock cycle the bus contains a single value of a fixed type, but the type of the value can change from clock cycle to clock cycle. The information indicating which type the value has may not even be part of the value itself. The type might instead be communicated in a separate signal, or have been sent on the bus on an earlier clock cycle. Currently a Hawk implementation of such a bus would require the designer to create a new datatype containing a constructor for each type of value the bus may transmit. The decision as to which constructor to use in a given clock cycle would have to be gleaned from whatever source the type information is being communicated, even if that source is in another module. The abstract Hawk designs we have created so far have not exhibited these problems, but it may become an issue when trying to model industrial microarchitectures "wire accurately".

7.1.5 Nondeterminism

The functional basis of Haskell causes Hawk circuits to be completely deterministic. For any fixed set of inputs, a Hawk circuit will always evaluate to the same value. In contrast, several hardware and concurrency oriented specification languages such as IOA[51], Ruby[39], SMV[58], and TLA+[41] allow circuits to have nondeterministic behaviors.

Nondeterministic circuits can be used to model partial specifications. For instance,

[1]Haskell implements typecasting through a (currently experimental) *universal type* mechanism.

the Hawk reference microarchitecture processes exactly one instruction per cycle, without stalling. One could instead model a reference processor that nondeterministically stalls zero or more clock cycles after processing an instruction. Any correctly designed pipelined processor would exhibit a set of behaviors[2] contained in the set of all reference processor behaviors. The pipelined machine's behaviors would then be said to *refine* the reference machine's behaviors.

The uniform treatment of specifications and implementations enabled by nondeterminism is considered by some to be an advantage of refinement-oriented specification languages. Hawk does not support nondeterminism directly, but we can simulate nondeterministic behaviors in Hawk through *oracles*, which are simply external parameters indicating which nondeterministic choice to make. For example, the *stuttering nats* circuit outputs the natural numbers in sequence, with possibly repeated elements:

$$
\begin{aligned}
& \mathit{stutterNats} \;::\; \mathit{Signal\ Bool} \to \mathit{Signal\ Int} \\
& \mathit{stutterNats}\ \mathit{stutter} \;=\; \mathit{out} \\
& \quad \mathit{where} \\
& \qquad \mathit{out} \;=\; \mathit{delay}\ 0\ (\mathit{select}\ \mathit{stutter}\ \mathit{out}'\ (\mathit{lift}\ (+\,1)\ \mathit{out})) \\
& \qquad \mathit{out}' \;=\; \mathit{delay}\ 0\ \mathit{out}
\end{aligned}
$$

The *stutter* parameter is a boolean signal indicating when to repeat the current value on the next clock cycle. By varying the values of *stutter* we can simulate all of *stutterNats* intended nondeterministic behaviors. If we wanted, we could go on to prove that *constant* 0 and the non-stuttering circuit *nats* both refine *stutteringNats*, by providing signals *constant True* and *constant False*, respectively, as the witness oracles. That is, we prove that

$$\mathit{stutteringNats}\ (\mathit{constant\ True}) \;=\; \mathit{constant}\ 0$$

and that

$$\mathit{stutteringNats}\ (\mathit{constant\ False}) \;=\; \mathit{nats}$$

[2] A pipelined processor specification could also have nondeterministic components, such as the latency of execution units or caches

We can also use oracles to show that one nondeterministic Hawk circuit refines another. For example, the circuit created by initially outputting 0 and then outputting the results of *stutteringNat* delayed by one cycle, is a refinement of the original *stutteringNat* circuit. We state this formally by existentially quantifying the oracle parameter:

$$\begin{array}{l} \forall\, \mathit{oracle}.\ \exists\, \mathit{oracle}'. \\ \quad \mathit{delay}\ 0\ (\mathit{stutteringNats}\ \mathit{oracle})\ =\ \mathit{stutteringNats}\ \mathit{oracle}' \end{array}$$

We can prove this law by choosing *oracle* arbitrarily and then supplying a witness oracle expression for oracle' in terms of *oracle*. The witness oracle we need to choose in this case is *delay True oracle*.

In general, a separate oracle parameter must be created for every independent source of nondeterminism in a circuit. This can become tedious for large, hierarchically specified circuits such as microarchitectures, and can make higher levels of the hierarchy hard to read[3].

On the other hand, a designer can explicitly create oracles in Hawk to exhibit specific (and repeatable) nondeterministic behaviors of interest. In particular, a designer can create *executable refinement mappings* that test whether one hawk circuit refines another. The designer first creates an oracle witness function in Hawk, then randomly generates a series of oracles. The implementation circuit is simulated on each randomly generated oracle, and the specification circuit is simulated on the oracle produced by invoking the witness function on the randomly generated oracle. If the witness function and the circuits are correctly written, the two circuits should output the same signals.

A designer can even run these tests before a formal refinement verification is carried out. Once the witness function has been thoroughly tested, it can be used directly in the refinement proof. In this way design engineers can assist in formal refinement verifications without having to become expert in the verification tools.

[3]This can be ameliorated to some extent by passing a record of oracles as a single parameter, or through the use of *implicit parameters*[47], an experimental Haskell feature for implementing dynamically scoped variables.

7.2 Transactions

Another major thrust of this thesis is the use of transactions as the central unit of communication between microarchitecture components. The notion of transactions as an abstract data type is independent of any specific hardware design language, although Hawk's support for structured datatypes and polymorphism make the concept easier to express.

7.2.1 Verifying pipelines with transactions

One of the transaction structure's major design benefits is its ability to express component interfaces uniformly, allowing designers to quickly interconnect microarchitecture subsystems at the block-diagram level. Another is the fact that the logic controlling a microarchitectural feature can usually be expressed in the component containing the data being controlled.

In our experience these advantages have been crucial to discovering algebraic laws. Take for example the bypass laws. If we were to express the *bypass* and *delay* components used in this law directly at the word level then there would have been a natural temptation to consolidate the logic controlling the bypass circuitry at the beginning of the pipeline, when the source register names first become available (because then only a couple of bits containing the results of the register name comparisons would have to be stored and sent to the bypass selection circuits, rather than the several dozen bits currently needed to send the register names themselves. Also, the logic needed to test whether the destination register was R0 would not have been duplicated in each bypass component.).

Unfortunately, this premature commitment to implementation efficiency can substantially complicate law discovery. We were not able to find the bypass laws until the "extra" control logic was localized to the data it was manipulating. Equally important was the reduction in the number of top-level pipeline components enabled by transactions. Components that are widely separated at the word level, such as the *kill* circuitry and the last *bypass* circuit in the pipeline of Chapter 3 appear much closer when expressed as transaction processors. This extra concision in specification made it easier to discover the hazard - bypass law, which spans multiple pipeline stages.

7.2.2 Calculating space efficient pipelines

Transactions help in quickly prototyping processor microarchitectures and significantly aid algebraic reasoning. However, directly synthesizing a transaction-processing microarchitecture to silicon would result in a circuit containing many unnecessary wires and state-holding elements, especially in later pipeline stages. We have performed some initial experiments on transformations that remove this unnecessary structure.

The idea is to define each microarchitecture component in terms of a *core* circuit and a *wrapper* circuit. The core circuit implements the component's functionality. The wrapper circuit is responsible for extracting the necessary transaction fields to deliver to the core, and packaging the results back up again as an output transaction. Transaction fields not needed by the core are passed through unmodified.

Microarchitecture synthesis then proceeds by expanding the pipeline's components into their constituent wrapper and core circuit definitions. A backwards dependency analysis on the pipeline's output wires determines which core components are actually used. The rest are unneeded and can be removed. A separate phase performs retiming and common subcircuit analysis to eliminate duplicate components. An interesting future research project would be to find out how efficient such a synthesized pipeline is relative to a pipeline designed entirely at the word level.

7.3 Algebraic reasoning

Hawk is designed to be a language that supports high level reasoning as well as specification. The algebraic reasoning developed in this thesis can be stratified into two essentially separate tasks: Proving the local microarchitecture component laws, and simplifying pipelines using those local laws.

7.3.1 Proving the component laws

Currently component law proofs seem to require quite a bit of verification expertise. A typical proof must perform induction over time and one or more case analyses on key transaction field values. Components can have large or unbounded state spaces, making completely automated techniques like model checking infeasible. Often the definition of a component needs to be generalized so that all values stored in internal *delay* circuits become parameters or return values. This was seen in Section 6.6.1 when we had to define the *rf* component in terms of a more general *envs* component that returned the entire contents of the register file at each clock cycle.

In some ways it is disappointing that components have to be so carefully constructed in order to get inductive proofs to succeed. Often the generalized components start looking like the state machine transducers common to more imperative specification languages. Originally this seemed to be a disadvantage of Hawk's stream-transformer style of specification, but we now tend to think of it as a general problem in theorem proving. It is often the case that recursively defined functions over an inductive domain have to be generalized to prove properties of interest. This is true regardless of the inductive domain. For multi-parameter functions the generalized form depends on which parameter is being inducted over and is thus an artifact of the proof, not the definition.

We speculate that in many cases this generalization step can be automated, provided the user specifies which parameter to induct over. For example, it should be possible to write an Isabelle tactic that automatically converts a first-order Hawk circuit description into a state-machine transducer form, even if the circuit contains occurrences of other recursively-defined circuits. If indeed such a tactic could be built then Hawk specifications could be written in a more natural style, and converted only as necessary for temporal induction proofs.

Higher order Hawk definitions are more of a challenge. There may not be an automatic way to convert a function that recurses over more than one parameter, as higher order components often do. In these cases the user would have to provide a conversion manually, which would then be used by the automated tactic when translating first order circuits containing the higher order component.

Typed versus untyped verification logics

Component proofs were also complicated by the type discipline imposed by higher order logic. We often wanted to quantify over elements of disparate types, particularly transaction field values. The transactions considered in this thesis had three types of fields: Register names, words, and opcodes. More sophisticated microarchitectures could have many more, such as exception flags, predication bits, thread identifiers, etc. It is a hassle having to create "universal" datatypes to inject these values into and coercion functions to move back and forth between them. An excellent article by Lamport and Paulson[42] discusses similar such problems. They suggest that although typed programming languages offer significant advantages, typed *specification* languages may not be the best choice, at least when it comes to carrying out formal correctness proofs. They make the following points (among others):

- Untyped set theory is an extremely expressive formalism, and underlies most of conventional mathematics.
- Simple type systems, such as the type system of higher order logic, significantly restrict the class of allowable specifications.
- Specifications containing "type errors" in untyped formalisms are quickly detected when attempting correctness proofs.
- Features found in more complex type systems such as predicate subtyping usually make type checking undecidable and can make it hard to modularize specifications. None of them approaches the flexibility of untyped set theory.
- In a mechanical verification system, a typed formalism automates routine inferences such as "if `x` is a `nat` and `y` is a `nat`, then `x + y` is a `nat`. However in a programmable theorem prover like Isabelle's ZF set theory logic these kinds of inferences can automated by writing a special "type-inference" tactic over the domain of the specification problem.

- Most formal specifications are not formally verified. Mechanical type checking can help catch errors in these cases.

They go on to observe that perhaps the best approach is to create an untyped specification language with the ability to build domain-specific type systems at the user level. Specifications could then be annotated and type-checked according to whatever type system is appropriate for that domain. Since the underlying formalism is untyped, specification fragments annotated using different type systems could be combined. Bogus type errors could be resolved during formal verification.

Our own experience with formally verifying typed Hawk specifications accords with their observations. Of course, it is easy to take for granted those things higher order logic does well and remember only the difficulties. We plan to re-verify some of the component laws in Isabelle's set theory formalism to get a more realistic sense of the tradeoffs involved.

Hawk is both a programming language and a specification language. Even within the Hawk team many more microarchitectures have been specified and simulated in Hawk than have been verified. So, on balance, strong typing has been a definite win. However, there is nothing preventing us specifying and simulating Hawk circuits in a typed language like Haskell, and then verifying them in an untyped formalism. Translation between Hawk and set theory could be automated, and inferred types can become set-membership constraints in the translated formalism.

7.3.2 Simplifying the pipeline

Currently, proving local component laws requires substantial experience in logic and inductive proof methods. Fortunately using the laws to simplify pipelines requires far less training in formal methods. As we demonstrated in Chapter 3, pipeline simplifications can be carried out graphically without resort to complex higher order reasoning or inductive generalization. Microarchitects quickly understand simplifications we present, and state that they would feel comfortable in applying the technique themselves. In fact, the Hawk team has considered building a *visual theorem prover* that would allow designers to carry out simplifications by selecting components and choosing from a menu of allowed transformations.

We have also found the microarchitecture laws to be fairly reusable when simplifying variations in the pipeline's design. We originally verified much simpler pipelines than the one presented in Chapter 3. Over time we increased the sophistication of the pipelines, and had to discover and prove new microarchitecture laws for the added components. Previously discovered laws, however, still remained applicable for the most part. This was true even when we added new transaction fields, such as the fields for carrying out branch speculation.

It remains to be seen how many of the component laws will still apply when simplifying more dynamic processor microarchitectures, such as those employing out of order execution and superscalar instruction fetching. In the microarchitectures presented in this thesis, the possible paths a transaction can take through a pipeline are limited, and closely correspond to the pipeline's component structure. In contrast, the paths a transaction takes through a modern out of order microarchitecture are much more determined by the structure of the program being executed than the pipeline. It is unclear whether structural simplification techniques will be as effective on such data-driven processors.

7.4 Converging equivalence relations

The CER framework was developed in this thesis to solve a specific problem – showing that recursively defined signals are well formed. Over time it has become clear that the technique can be generalized to solve a wide range of recursive equations outside of the context of Hawk, such as the noncomputable function definitions for filtering and flattening infinite lists. In fact, as shown in Section 5.8 a CER combinator can be defined that is powerful enough to define any well-founded recursive function.

It would be quite interesting to build a recursive function definition package based on the CER framework. The package would be as expressive as existing packages for well-founded functions, but could also define non-well-founded functions as well.

7.5 Mechanizing the verification

We were able to automate within Isabelle many, but not all, of the paper-and-pencil proofs performed for this work. Specifically, we successfully generated Isabelle proofs for the following theories and components:

- **Converging Equivalence Relations theory.** This theory includes proofs of the CER fixpoint theorem, the signal and lazy list CER axioms, and the CER combinators. A descendant theory proves that the recursive equations defining the functions *iterates*, *lmap*, *lappend*, *lfilter*, and *lflatten* have unique solutions.

- **Recursive Hawk circuit definitions.** Several Hawk circuits containing feedback signals are defined in Isabelle by invoking the CER fixpoint theorem, including the resettable counter circuit of Section 2.1, the *envs* circuit of Section 6.6.1, and several pipelined microarchitectures.

- **Microarchitecture component laws.** Most of the time-invariance laws have Isabelle proofs, as well as the feedback rotation law, the register file - bypass law, and the hazard - bypass law. We did not have time to prove the laws governing the `no_haz` and `branch_misp` components. However, their proofs should not present any difficulties, now that a theory of first class field names has been developed (see below).

- **First class field names theory.** The theory of transaction field names has only recently been mechanized in Isabelle. The primary motivation for mechanizing this theory is to make the microarchitecture component law proofs more robust in the face of changes to the transaction datatype. Previously, whenever the transaction type was extended by new field declarations, substantial portions of the component law proofs would have to be modified. It is, in fact, the main reason why the `no_haz` and `branch_misp` laws have not yet been mechanized, since these components require the addition of the **specPCFld** transaction field (which is not currently part of the transaction datatype). Now that disparately-typed fields can be quantified over, adding the **specPCFld** should be a much simpler matter.

- **Pipeline simplification theory.** To verify the top-level pipeline simplification presented in Chapter 3 we axiomatized all of the microarchitecture laws in a separate theory, and then used the laws as rewrite rules to simplify the pipelined microarchitecture of Figure 3.10 to the reduced microarchitecture of Figure 3.67. We overcame difficulties with Isabelle's rewriting tactics by converting the microarchitecture laws and pipeline to a different form, described in Section 7.5.2.

7.5.1 Mechanizing the microarchitecture law proofs

The paper-and-pencil proofs of microarchitecture laws can be quite lengthy, even for simple circuits such as the registerFile-bypass law. Fortunately many of the steps simply consist of rewriting with respect to previously proven theorems. These steps can be automated in theorem provers like Isabelle that can repeatedly simplify a subgoal with respect to a list of equational theorems.

A more difficult part of proving microarchitectural laws is defining stateful components in terms of appropriate auxiliary functions. The auxiliary functions of a component need to be defined in such a way that all of the component's important internal states are visible during the inductive proof. For the registerFile-bypass law, this involved defining the auxiliary function *envs*, which exposed the internal state of the register file contents. It turns out that in many cases these auxiliary functions are essentially the component's corresponding state machine transducers, as is the case for *envs*. The benefit of such transducer-like functions is that one can relate the value returned by the transducer in the next clock cycle in terms of the inputs to the transducer in the next clock cycle and values returned by the transducer in the current clock cycle. This is precisely the form of relation needed when carrying out temporal induction.

7.5.2 Mechanizing the top level pipeline simplification

Simplifying microarchitectures algebraically in Isabelle has been problematic. Hawk pipeline definitions consist of mutually-recursive signal definitions. Isabelle's rewriting tactics can handle mutually-recursive pattern matching function definitions by only rewriting functions that are applied to explicit constructors. This is exactly what is needed to prove the inductive properties used in the component laws, and Isabelle's sophisticated conditional and higher order rewriting package is of great help there. Unfortunately Hawk signal definitions at the pipeline level do not use pattern matching, and so naive rewriting often loops. Instead, top level rewrite steps must currently be done one at a time.

Another difficulty concerns expression sharing. In Isabelle let-expressions are just syntactic sugar for function applications. Most Isabelle tactics do not support let-expressions directly, requiring the user to expand them first. The problem with this is that all sharing of sub-terms is lost during the expansion. Hawk microarchitectures contain significant signal sharing, and expanding them can increase the size of a pipeline by an order of magnitude. This size increase also increases tactic execution time by an order of magnitude, and makes the pipeline much harder to read during verification.

One possible solution to both of these problems is to add support in Isabelle for recursive let-expressions (letrecs), defined as unique fixed points. When simplifying letrecs, later variable declarations could be rewritten in terms of earlier declarations automatically, but not vice-versa. Special tactics could be defined to change the order of declarations in a letrec if a different rewriting order was desired. An earlier declaration would only be expanded in a later declaration if it then enabled a rewrite rule to simplify the expanded expression. New common subterms created during simplification would be collected as shared variable declarations.

Adding letrec support would not require changing Isabelle's trusted kernel of primitive inference rules. It would require substantially modifying tactics written outside of the kernel, such as the rewriter and tableau resolution tactics. Instead, we followed an alternative approach that reuses more of Isabelle's existing infrastructure.

Conversion to relational form

To take advantage of Isabelle's existing tools for reasoning about formulas (terms of type *bool*), the microarchitecture laws and pipeline definition were first converted to *relational form.* A Hawk circuit in relational form is represented as a predicate equality, rather than a function. The equality is parameterized on both the input and the output signals of the circuit, by representing the signals as free variables. The predicate equation is true exactly when the outputs equal the result of applying the circuit to the inputs. For example, the relational form of the *regFile* circuit could be given in terms of the free output variable *out*, and the free input variables *inp* and *rb*:

$$out = regFile\ inp\ wb$$

More complex circuits containing internal signals and recursion can be expressed in relational form through the use of existential quantification and conjunction. For instance, recall the register file - bypass law, presented again in Figure 7.1 with named internal wires.

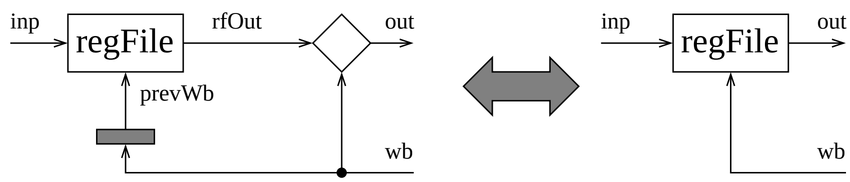


Figure 7.1: register file - bypass law

The circuit on the left hand side of this law can be expressed as the following relation on the free variables *out*, *inp* and *wb*:

$$\begin{aligned}(\exists\ prevWb\ rfOut.\ prevWb &= delay\ nop\ wb \wedge \\ rfOut &= regFile\ inp\ prevWb \wedge \\ out &= bypass\ rfOut\ wb)\end{aligned}$$

The internal signal variable names *prevWb* and *rfOut* are bound by the existential quantifier, and thus are not visible in any enclosing context. Specifying circuits as relations in this way is a commonly taken approach when verifying hardware designs in higher order logic[28, 64].

The microarchitecture law of Figure 7.1 can now be expressed in higher order logic as an equality between the two circuit relations:

$$
\begin{aligned}
&\forall\ out\ inp\ wb.\\
&\quad (\exists\ prevWb\ rfOut.\ prevWb = delay\ nop\ wb \wedge\\
&\qquad\qquad\qquad\qquad rfOut = regFile\ inp\ prevWb \wedge\\
&\qquad\qquad\qquad\qquad out = bypass\ rfOut\ wb)\\
&\quad =\\
&\quad (out = regFile\ inp\ wb)
\end{aligned}
$$

Assuming that the pipeline to be simplified is also expressed in relational form, then the above equality (once proven) can be used as a rewrite rule, at least in principle. In practice, Isabelle's current rewriting tactics are too restrictive to use such rules naturally. In particular, the rewriting tactics will not apply existentially-quantified rewrite rules unless the order of the existential quantifiers in the left hand side of the rewrite rule exactly matches the order in the term being rewritten. Similarly, the order of each conjunct in the rewrite rule must exactly match the order of the subject term's conjuncts.

To remedy this situation we developed a set of tactics that allow conjuncts and existential variables within Isabelle to be reordered on demand. We also developed tactics to apply the circuit duplication and feedback - rotation laws in this relational setting. Once the tactics were written, the pipeline was simplified step by step. Unfortunately, each step had to be carried out manually, requiring a total of 88 tactic invocations. However, there is nothing in principle to prevent Isabelle's rewriting tactics from being generalized to handle existentially quantified conjunctions in rewrite rules. Upgrading the rewriting tactics would dramatically reduce the number of manual simplification steps required.

Justifying the conversion to relational form

The question still remains as to whether relational conversions are valid, as currently Hawk microarchitecture laws are verified as equations between values, not relations. Fortunately the conversion can be justified by the fact that all recursive Hawk circuits are defined as unique fixed points.

To begin with, any recursive circuit in Hawk can be expressed as a function of a projection p and a unique fixed point $\mathit{fix}\ F$

$$circuit \equiv \lambda inputs.\ p\ (fix\ F)$$

by expressing the input signals of the circuit as parameters of the function (i.e. *inputs*), and both the output and internal wires of the circuit as elements of a tuple, which becomes the result of *fix F*. The function p then projects out only the output signals.

For example, the circuit shown in the left hand side of Figure 7.1 can be expressed as

$$\begin{array}{l}
\lambda\ inp\ wb.\ p\ (fix\ F) \\
\text{where} \\
\quad F \ = \ (\lambda\,(out, prevWb, rfOut)\ .\ (\ bypass\ rfOut\ wb, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad delay\ nop\ wb, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad regFile\ inp\ prevWb)) \\
\quad p \ = \ (\lambda\,(out, prevWb, rfOut)\ .\ out)
\end{array}$$

Thus, any microarchitectural law can be written as an equation between two circuits of the form:

$$(\lambda i_1 \ldots i_n.\ p_1\ (\text{fix}\ F_1)) = (\lambda i_1 \ldots i_n.\ p_2\ (\text{fix}\ F_2))$$

where F_1 and F_2 may contain occurrences of $i_1 \ldots i_n$. If F_1 and F_2 have unique fixed points (possibly over different types), then the above equation is provably equivalent in higher order logic to

$$\begin{array}{l}
\forall\ i_1\ \ldots\ i_n\ out. \\
\quad (\exists\ tuple_1.\ tuple_1 \ = \ F_1\ tuple_1\ \wedge\ out \ = \ p_1\ tuple_1) \\
\quad\ = \\
\quad (\exists\ tuple_2.\ tuple_2 \ = \ F_2\ tuple_2\ \wedge\ out \ = \ p_2\ tuple_2)
\end{array}$$

When this equation is expanded in terms of F_1, F_2, p_1, and p_2, and simplified by the following tuple equality rule

$$\forall x_1\ x_2\ y_1\ y_2.\ ((x_1, y_1) = (x_2, y_2)) = (x_1 = x_2 \wedge y_1 = y_2)$$

the resulting reduced equation is in the required relational form.

Performing the same inferences in reverse order, two circuits that have been proven equivalent in relational form can be converted to an equality between the same circuits expressed as unique fixed points. Thus microarchitecture pipelines that have been simplified relationally can be converted back into conventional Hawk expression form.

It is important to note that conversion to relational form may not be valid in general when F_1 and F_2 do not have unique fixed points.

7.6 Conclusions and further research directions

In all, Hawk has proved to be an excellent platform for quickly specifying and reasoning algebraically about pipelined microarchitectures at an abstract level. The strengths of Hawk revolve around its abstraction capabilities and executability:

- **Abstract and modular specification.** The combination of functional language structuring principles with the domain-specific transaction ADT leads to remarkably concise, yet understandable, pipelined microarchitecture descriptions. In particular, transactions combined with mutual recursion at the stream level allow us to build processor components as separate modules, then easily compose them at the top level.

- **Abstract and modular reasoning.** At the same time, the simple semantics underlying Hawk allows one to reason about source level hawk descriptions directly as expressions in higher order logic. The equational laws we have derived for local microarchitecture components are independent of context, and can thus be used in a modular fashion. Hawk's equational theorems and proofs can also be displayed visually, so that users do not need to be versed in the complexities of higher order or temporal logic to follow them.

- **Executability.** Hawk is fully executable, so designers can test new designs on concrete and symbolic inputs. One can even simulate first order Hawk microarchitectures visually. Hawk project members Thomas Nordin and Byron Cook have developed *Visual Hawk*, a graphical front end to the Hawk interpreter. In Visual Hawk a designer can create circuit diagrams by dragging microarchitecture components from a palette onto a canvas and then connecting them with wires. Each wire represents a signal. The tool performs static type checking and input/output mode analysis to ensure that wires are only connected between compatible component ports. The designer can simulate microarchitecture circuits interactively, and then double click on a wire to obtain a trace of all the values sent along it so far.

The executability of Hawk combined with good user interface support makes Hawk a useful tool to designers even in the absence of formal verification.

- **Embedded language.** Hawk is built upon and compatible with the general-purpose programming language Haskell. Thus we immediately can make use of the existing interpreters, compilers, programming texts, and user community associated with Haskell.

Of course, Hawk is not perfect. The major weaknesses of Hawk and the algebraic method uncovered during the course of this thesis involve simulation efficiency and lack of automation when verifying Hawk circuits:

- **Efficiency.** Hawk's high level of abstraction comes at a cost. Current Hawk implementations run at two to three orders of magnitude more slowly than state of the art imperative microarchitecture simulators. Much of this slowdown comes from using a general purpose Haskell compiler, and the efficiency of Hawk simulations could be improved by at least two orders of magnitude by employing domain-specific compilation techniques, such as converting streams into mutable variables, statically scheduling expression evaluation, monomorphizing polymorphic expressions, and custom garbage collection. But it is unclear even with these optimizations how closely we could approach the efficiency of the very best hand-tuned microarchitecture simulators.

- **Infinite state spaces.** Most Hawk components operate over unbounded datatypes. For instance, the register file component can have an infinite number of registers.

Each register can contain a word value of unbounded size. Similar generality is built into other components, such as the instruction and data caches. Unfortunately, most of the fully automatic model checking algorithms operate over finite state spaces, and thus can not be used directly. Theorem provers can easily handle infinite state spaces, but require a great deal of effort and expertise to use. In practice, this significantly limits the size of Hawk specifications that can be verified in a reasonable amount of time. However, a promising intermediate technology called *compositional model checking* (discussed in the future work section below) may help reduce the amount of manual intervention needed to prove microarchitecture laws.

- **Hidden state.** At the lowest level, local equational laws have to be proved by some form of induction. Often one has to generalize the equation being proved to a bisimulation relation that holds at all points in time, and relates values at the previous time step to values at the current time step. With state machine formalisms, all of these previous and current values can be referenced explicitly. Hawk components, on the other hand, tend to hide previous values (which are the outputs of *delay* circuits) deep within the component definitions. To build a suitable bisimulation in Hawk, one has to parameterize initial arguments to *delay* circuits as arguments to the entire component, or one has to construct auxiliary state observation functions and define the bisimulation in terms of these. In either case it is extra work that does not need to be done with state machine oriented verification.

From a generic theorem proving perspective this thesis' most widely applicable result is the development of converging equivalence relations. The CER framework generalizes definition by well-founded recursion, and may turn out to be a useful way to define functions over a broad range of coinductive data structures, such as infinite lists, infinite trees, and cyclic graphs.

Future work

The work described here can be extended in several directions. Besides increasing the power of Isabelle by adding support for recursive let-expressions and CER-definable functions, we also intend to complete the algebraic verification of the pipeline of Chapter 3. At the moment the pipeline cannot be simplified to a reference machine because of the extra *nop* transactions output when the pipeline stalls. It should however be possible to simplify the pipeline to a stalling reference machine, where the reference machine's stalling behavior is governed by an external oracle. By feeding the stalling control logic of the pipelined machine as the reference machine's oracle, the two microarchitectures should output exactly the same transactions. Given suitable additional component laws concerning the instruction cache, it should be possible to prove the two processors equivalent algebraically.

We also intend to automate component law proofs further by applying recent work on *abstract model checking*, particularly the work of McMillan[57, 61, 59, 60] on verifying infinite state models. McMillan and the author have performed some preliminary experiments on verifying component laws, with promising results. Abstract model checking was able to reduce infinite state space versions of the registerFile - bypass and hazard - bypass laws down to a series of small finite state model checking problems, which were then solved automatically. It was necessary to add a few refinement maps and manual annotations stating which variables were used in a symmetric manner, but overall the approach seemed much more automatic than the current inductive proofs carried out in Isabelle.

Bibliography

[1] Mark Aagaard and Miriam Leeser. Reasoning about pipelines with structural hazards. In *Theorem Provers in Circuit Design, TPCD'94* (Bad Herrenalb, Germany, 1994), LNCS, v. 901, pages 13–32. Springer-Verlag, New York, 1995.

[2] Mark D. Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-Johan H. Seger. A methodology for large-scale hardware verification. To appear in proceedings of *Formal Methods in Computer Aided Design, FMCAD'00*, 2000.

[3] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: A pragmatic implementation of combined model checking and theorem proving. In *Theorem Proving in Higher Order Logics, TPHOLs'99* (Nice, France, 1999), LNCS, v. 1690, pages 323–340. Springer, New York, 1999.

[4] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, San Francisco, 1996.

[5] Craig Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In *Formal Methods in Computer Aided Design, FMCAD'96* (Palo Alto, 1996), LNCS, v. 1166, pages 187–201. Springer, New York, 1996.

[6] David Barton. Advanced modeling features of MHDL. In *International Conference on Electronic Hardware Description Languages, ICEHDL'95* (Las Vegas, 1995), pages 71–77. Society for Computer Simulation, 1995.

[7] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, Englewood Cliffs, 1988.

[8] Richard S. Bird and Oege De Moor. *Algebra of Programming*. Prentice Hall, Englewood Cliffs, 1996.

[9] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *The 1998 ACM SIGPLAN International Conference on Functional Programming, ICFP'98* (Baltimore, Maryland, 1998), pages 174–184. ACM Press, New York, 1998.

[10] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *IFIP Transactions on Theorem Provers in Circuit Design* (Nijmegen, The Netherlands, 1992), pages 129–156. North-Holland, New York, 1992.

[11] Jerry Burch and David Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification, CAV'94* (Stanford, 1994), LNCS, v. 818, pages 68–80. Springer, New York, 1994.

[12] Girard Buskes and Arnoud van Rooij. *Topological Spaces: from distance to neighborhood.* Springer, New York, 1997.

[13] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Advances in Computing Science - ASIAN'99* (Phuket, Thailand, 1999), LNCS, v. 1742, pages 62–73. Springer, New York, 1999.

[14] Koen Claessen and Mary Sheeran. *A tutorial on Lava: A hardware description and verification system.* Revised: April 7th, 2000. Available: `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/svh` [Viewed: August 9th, 2000].

[15] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Computer Aided Verification, CAV'93* (Elounda, Greece, 1993), LNCS, v. 697, pages 450–462. Springer-Verlag, New York, 1993.

[16] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[17] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, Cambridge, 1999.

[18] Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. *Workshop on Formal Techniques for Hardware and Hardware-like Systems, FTH'98* (Unpublished, Marstrand, Sweden, 1998). Available: `http://www.cse.ogi.edu/PacSoft/projects/Hawk/papers` [Viewed: August 10th, 2000].

[19] Nancy Day, Jeffrey Lewis, and Byron Cook. Symbolic simulation of microprocessor models using type classes in Haskell. Technical Report CSE-99-005, Oregon Graduate Institute, Computer Science Department, Portland, Oregon, 1999.

[20] Nancy A. Day, Mark D. Aagaard, and Byron Cook. Combining stream-based and state-based verification techniques for microarchitectures. To appear in proceedings of *Formal Methods in Computer Aided Design, FMCAD'00*, 2000.

[21] Marco Devillers, David Griffioen, and Olaf Müller. Possibly infinite sequences in theorem provers: A comparative study. In *Theorem Proving in Higher Order Logics, TPHOLs'97* (Murray Hill, New Jersey, 1997), LNCS, v. 1275, pages 89–104. Springer, New York, 1997.

[22] Keith Devlin. *The Joy of Sets: Fundamentals of Contemporary Set Theory.* Springer-Verlag, New York, 1993.

[23] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming, ICFP'97* (Amsterdam, The Netherlands, 1997), pages 263–273. ACM Press, New York, 1997.

[24] E. Allen Emerson and Richard J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Correct Hardware Design and Verification Methods, CHARME'99* (Bad Herrenalb, Germany, 1999), LNCS, v. 1703, pages 142–156. Springer-Verlag, New York, 1999.

[25] Jacob Frost. A case study of co-induction in Isabelle. Technical Report 359, University of Cambridge, Computer Laboratory, February 1995. Revised version of CUCL 308, August 1993.

[26] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *Functional Programming Languages and Computer Architecture, FPCA'93* (Copenhagen, Denmark, 1993), pages 223–232. ACM Press, New York, 1993.

[27] M. J. C. Gordon. From LCF to HOL: a short history. Revised: 2000. Available: `http://www.cl.cam.ac.uk/users/mjcg/papers/HolHistory.html` [Viewed: August 10th, 2000].

[28] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. North-Holland, New York, 1986.

[29] M. J. C. Gordon and eds. T. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, New York, 1993.

[30] Mike Gordon. The semantic challenge of Verilog HDL. In *Logic in Computer Science, LICS'95* (San Diego, 1995), pages 136–145. IEEE Computer Society Press, Los Alamitos, 1995.

[31] David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer Aided Design, FMCAD'98* (Palo Alto, 1998), LNCS, v. 1522, pages 321–333. Springer, New York, 1998.

[32] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, Cambridge, 1992.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann, San Francisco, 1995.

[34] Paul Hudak and Mark Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ...: An experiment in software prototyping productivity. Technical Report YALEU/DCS/RR-1049, Yale University, October 1994.

[35] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell, version 98. Revised: June 2000, by Reuben Thomas. Available: `http://www.haskell.org/tutorial` [Viewed: August 10th, 2000].

[36] Steven D. Johnson. *Synthesis of Digital Systems from Recursive Equations.* MIT Press, Cambridge, 1984.

[37] G. Jones and M. Sheeran. The study of butterflies. Technical Report PRG-TR-14-90, Programming Research Group, Oxford University Computing Laboratory, 1990.

[38] Geraint Jones and Mary Sheeran. Timeless truths about sequential circuits. In *Concurrent Computations: Algorithms, Architectures and Technology.* Plenum Press, New York, 1988. Available: `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Geraint.Jones/MUFP-1-87.ps.Z` [Viewed: August 10th, 2000].

[39] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In *Formal Methods for VLSI Design*, pages 13–70. North-Holland, New York, 1990.

[40] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Mathematics of Program Construction*, LNCS, v. 669, pages 107–136. Springer-Verlag, New York, 1993.

[41] Leslie Lamport. Specifying concurrent systems with TLA+. Revised: March 3rd, 1999. Available: `http://www.research.digital.com/SRC/tla/papers.html` [Viewed: August 10th, 2000].

[42] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.

[43] John Launchbury. Graph algorithms with a functional flavour. In *Advanced Functional Programming*, LNCS, v. 925, pages 308–331. Springer, New York, 1995.

[44] John Launchbury et al. Hawk release 2.2. Revised: November 22nd, 1999. Available: `http://www.cse.ogi.edu/PacSoft/projects/Hawk/` [Viewed: August 10th, 2000].

[45] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[46] Jeremy Levitt and Kunle Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *Design Automation Conference, DAC'96* (Las Vegas, 1996), pages 558–563. ACM Press, New York, 1996.

[47] Jeffrey Lewis, Mark Shields, Erik Meijer, and John Launchbury. Implicit parameters: Dynamic scoping with static types. In *Principles of Programming Languages, POPL'00* (Boston, 2000), pages 108–118. ACM Press, New York, 2000.

[48] Yanbing Li. HML: An innovative hardware design language and its translation to VHDL. Master's thesis, Cornell University, 1995.

[49] Yanbing Li and Miriam Leeser. HML: An innovative hardware design language and its translation to VHDL. In *IFIP International Conference on Computer Hardware Description Languages and their Applications, CHDL'95* (Tokyo, Japan, 1995), pages 691–696. IEEE, New York, 1995. Revised: June 18th, 1999. Available: `ftp://ftp.ece.neu.edu/pub/mel/hml/paper` [Viewed: August 10th, 2000]. ISBN: 4930813670.

[50] Carl Johan Lillieroth and Satnam Singh. Formal verification of FPGA cores. *Nordic Journal of Computing*, 6(3):299–319, October 1999.

[51] N. A. Lynch and M. R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[52] John Matthews. Recursive function definition over coinductive types. In *Theorem Proving in Higher Order Logics, TPHOLs'99* (Nice, France, 1999), LNCS, v. 1690, pages 73–90. Springer, New York, 1999.

[53] John Matthews and John Launchbury. Elementary microarchitecture algebra. In *Computer Aided Verification, CAV'99* (Trento, Italy, 1999), LNCS, v. 1633, pages 288–300. Springer, New York, 1999.

[54] John Matthews and John Launchbury. Elementary microarchitecture algebra: Top-level proof of pipelined microarchitecture. Technical Report CSE-99-002, Oregon Graduate Institute, Computer Science Department, Portland, Oregon, March 1999.

[55] John Matthews, John Launchbury, and Byron Cook. Microprocessor specification in Hawk. In *International Conference on Computer Languages, ICCL'98* (Chicago, 1998), pages 90–101. IEEE Computer Society, Los Alamitos, 1998.

[56] William McCune and Larry Wos. Otter—the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, April 1997.

[57] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Computer Aided Verification, CAV'98* (Vancouver, BC, Canada, 1998), LNCS, v. 1427, pages 110–121. Springer, New York, 1998.

[58] Ken L. McMillan. *Symbolic model checking.* Kluwer Academic Publishers, Boston, 1993.

[59] Ken L. McMillan. Circular compositional reasoning about liveness. Technical Report 1999-02, Cadence Berkeley Labs, Cadence Design Systems, 1999.

[60] Ken L. McMillan. A methodology for hardware verification using compositional model checking. Technical Report 1999-03, Cadence Berkeley Labs, Cadence Design Systems, 1999.

[61] Ken L. McMillan. Verification of infinite state systems by compositional model checking. Technical Report 1999-01, Cadence Berkeley Labs, Cadence Design Systems, 1999.

[62] T. F. Melham. Abstraction mechanisms for hardware verification. In *VLSI Specification, Verification and Synthesis*, pages 129–157. Kluwer Academic Publishers, Boston, 1988.

[63] T. F. Melham. Automating recursive type definitions in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, New York, 1989.

[64] T. F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, New York, 1993.

[65] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[66] Paul Miner. *Hardware Verification Using Coinductive Assertions*. PhD thesis, Indiana University, 1998.

[67] J. Strother Moore. Symbolic simulation: An ACL2 approach. In *Formal Methods in Computer Aided Design, FMCAD'98* (Palo Alto, 1998), LNCS, v. 1522, pages 334–350. Springer, New York, 1998.

[68] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[69] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle's logics: HOL. Revised: October 31st, 1999. Available: `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html` [Viewed: August 10th, 2000].

[70] John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *FPLE'95, Functional Programming Languages in Education*, LNCS, v. 1022, pages 195–214. Springer, New York, 1995.

[71] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal* [Online], 1999. Revised: First quarter, 1999. Available: `http://developer.intel.com/technology/itj/` [Viewed: August 18th, 2000]. (Note that this online journal does not have individual page numbers or give volume/issue number for this title).

[72] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages, ICCL'98* (Chicago, 1998), pages 80–89. IEEE Computer Society, Los Alamitos, 1998.

[73] Lawrence Paulson. *Isabelle: A Generic Theorem Prover*. LNCS, v. 828. Springer-Verlag, New York, 1994.

[74] Lawrence C. Paulson. The Isabelle reference manual. Revised: October 31st, 1999. Available: `http://www.cl.cam.ac.uk/Research/HVG/Isabelle` [Viewed: August 10th, 2000].

[75] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175–204, April 1997.

[76] Simon L. Peyton Jones et al. Haskell 98: A non-strict, purely functional language. Revised: February 1st, 1999. Available: `http://www.haskell.org/onlinereport` [Viewed: August 10th, 2000].

[77] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.

[78] Prover Technology AB. NP-tools 2.4: A commercial propositional validity checker and satisfaction algorithm. Revised: 2000. Available: `http://www.prover.com` [Viewed: August 10th, 2000].

[79] O. Rasmussen. Formalising Ruby in Isabelle ZF. In L. C. Paulson, editor, *Proceedings of the First Isabelle Users Workshop,* Tech. Report 379, pages 246–265. University of Cambridge, September 1995.

[80] Walter Rudin. *Principles of Mathematical Analysis.* McGraw-Hill, New York, third edition, 1976.

[81] John Rushby and David W. J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report SRI-CSL-95-10, SRI International, Menlo Park, CA, June 1995. Revised, July 1996.

[82] J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In *Computer Aided Verification, CAV'98* (Vancouver, BC, 1998), LNCS, v. 1427, pages 135–146. Springer, New York, 1998.

[83] J. Saxe and S. Garland. Using transformations and verifications in circuit design. *Formal Methods in System Design*, 4(1):181–210, 1994.

[84] R. Sharp and O. Rasmussen. An introduction to Ruby. Teaching Notes ID–U: 1995-80, Dept. of Computer Science, Technical University of Denmark, 1995.

[85] R. Sharp and O. Rasmussen. The T-Ruby design system. In *Computer Hardware Description Languages and their Applications, CHDL'95* (Tokyo, Japan, 1995), pages 587–596. 1995. Available: `ftp://ftp.it.dtu.dk/pub/Ruby/chdl95.ps.Z` [Viewed: August 10th, 2000].

[86] M. Sheeran. Retiming and slowdown in Ruby. In *The Fusion of Hardware Design and Verification* (Glasgow, Scotland, 1988), pages 289–308. North-Holland, New York, 1988.

[87] K. Slind. Derivation and use of induction schemes in higher-order logic. In *Theorem Proving in Higher Order Logics, TPHOLs'97* (Murray Hill, New Jersey, 1997), LNCS, v. 1275, pages 275–290. Springer, New York, 1997.

[88] Konrad Slind. Function definition in higher order logic. In *Theorem Proving in Higher Order Logics, TPHOLs'96* (Turku, Finland, 1996), LNCS, v. 1125, pages 381–398. Springer-Verlag, New York, 1996.

[89] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, April 1997.

[90] Robert D. Tennent. *Semantics of Programming Languages*. Prentice Hall, 1991.

[91] Norbert Völker. Disjoint sums over type classes in HOL. In *Theorem Proving in Higher Order Logics, TPHOLs'99* (Nice, France, 1999), LNCS, v. 1690, pages 5–18. Springer, New York, 1999.

[92] Philip Windley and Michael Coe. A correctness model for pipelined microprocessors. In *Theorem Provers in Circuit Design, TPCD'94* (Bad Herrenalb, Germany, 1994), LNCS, v. 901, pages 33–51. Springer-Verlag, New York, 1995.

Biographical Note

John Matthews was born in 1967, in Corvallis, Oregon. He received his B.S. degree in Computer Science, Math, and Statistics from the University of Washington in 1990. He then worked for five years at Hewlett-Packard Company as a software engineer. He returned to school at the Oregon Graduate Institute in 1995, pursuing a Ph.D. degree in Computer Science.