

Introducing the PilGRIM: A Processor for Executing Lazy Functional Languages

Arjan Boeijink, Philip K.F. Hölzenspies, and Jan Kuper

University of Twente
Enschede, The Netherlands
w.a.boeijink@utwente.nl

Abstract. Processor designs specialized for functional languages received very little attention in the past 20 years. The potential for exploiting more parallelism and the developments in hardware technology, ask for renewed investigation of this topic. In this paper, we use ideas from modern processor architectures and the state of the art in compilation, to guide the design of our processor, the PilGRIM. We define a high-level instruction set for lazy functional languages and show the processor architecture, that can efficiently execute these instructions.

1 Introduction

The big gap between the functional evaluation model, based on graph reduction, and the imperative execution model of most processors, has made (efficient) implementation of functional languages a topic of extensive research. Until about 20 years ago, several projects have been undertaken to solve the implementation problem by designing processors specifically for executing functional languages. However, advances in the compilation strategies for conventional hardware and the rapid developments in clock speed of mainstream processor architectures made it very hard for language specific hardware to show convincing benefits.

In the last 20 years, the processors in PCs have evolved a lot. The number of transistors of a single core has grown from hundreds of thousands to hundreds of millions following Moore’s law and the clock speed has risen from a few MHz to a few GHz [5]. Introduction of deep pipelines, superscalar, and out-of-order execution changed the microarchitecture of processors completely. Currently, processor designs are limited by power usage instead of transistor count, which is known as the *power wall*. Two other “walls” limiting the single thread performance of modern processors are the memory wall (memory latency and bandwidth bottleneck) and the Instruction Level Parallelism (ILP) wall (mostly due to the unpredictability of control flow). These three walls have shifted the focus in processor architectures from increasing frequencies to building multicore processors.

Recent work on the Reduceron [9] showed encouraging results demonstrating what can be gained by using low-level parallelism to design a processor specifically for a functional language. The emphasis of this low-level parallelism is on

simultaneous data movements, but also includes instruction level parallelism. Functional languages are well suited for applications that are hard to parallelize, such as complex symbolic manipulations. Single thread performance will be critical when all easy parallelism has been fully exploited. Laziness and first-class functions make Haskell very data and control flow intensive [10]. Unlike research done in previous decades, language specific hardware modifications are now bounded by complexity (of design and verification) and not the number of transistors.

Performance gains for specialized pure functional architectures might influence conventional processor architectures, because the trend in mainstream programming languages is towards more abstraction, first-class functions, and immutable data structures.

The changes in hardware technology, the resulting modern hardware architectures, and the positive results of the Reduceron suggest it is time to evaluate processors for functional languages again.

2 The Pipelined Graph Reduction Instruction Machine

In this paper, we introduce the PilGRIM (the name is an acronym of the section title). The PilGRIM is a processor with a design that is specialized for executing lazy functional languages. The architecture is derived from modern general purpose architectures, with a 64-bit datapath and using a standard memory hierarchy: separate L1 instruction and data caches, an L2 cache, and DDR-memory interface. The design targets silicon and is intended to be a realistic design for current hardware technology. The instruction set is designed to exploit benefits of extensive higher level compiler optimizations (using the output of GHC). This is especially important for performance of primitive (arithmetic) operations. The processor executes a high-level instruction set that is close to a functional core language and that allows code generation to be simple. The PilGRIM reuses some of the basic principles behind the Big Word Machine (BWM) [1] and the Reduceron [8].

We intend to pipeline this processor deep enough to make a clock frequency of 1GHz a feasible target. The details of the pipeline structure are work in progress. For space reasons, we will not discuss pipelining in this paper, but many design choices are made in preparation of a deep pipeline.

The contributions of this paper are:

- A high-level and coarse-grained instruction set for lazy functional languages, with a simple compilation scheme from a functional core language.
- Proposal of a hardware architecture that can efficiently execute this instruction set, where this design is made with a deep pipeline in mind.

3 Instruction Set and Compilation

Before designing the hardware, we first want to find a suitable evaluation model and an instruction set, because designing hardware and an instruction set simultaneously is too complex. We chose to use GHC as the Haskell compiler frontend,

because its External Core [15] feature is a convenient starting point on which to base a code generator for our new architecture. The extensive set of high-level optimizations provided by GHC is a big additional benefit.

While External Core is a much smaller language than Haskell, it is still too complex and too abstract for efficient execution in hardware. Before compiling to an instruction set, we transform External Core to a simplified and low-level intermediate language, defined in the next section. The instruction set and assembly language is derived from Graph Reduction Intermediate Notation (GRIN) [4,3] (an intermediate language, in the form of a first-order monadic functional language). GRIN has been chosen as our basis, because it is a simple sequential language, that has a small set of instructions and that is close to a functional language.

<p>function definition:</p> $d ::= f x^+ = e$ <p style="margin-left: 2em;"> $g = e$</p> <p>toplevel expression:</p> $e ::= s$ <table style="margin-left: 2em; border: none;"> <tr><td> let b in e</td><td><i>(simple expr.)</i></td></tr> <tr><td> letS b in e</td><td><i>(lazy let expr.)</i></td></tr> <tr><td> case s of $\{ a^+ \}$</td><td><i>(case expr.)</i></td></tr> <tr><td> if c then e else e</td><td><i>(if expr.)</i></td></tr> <tr><td> fix $(\lambda r. f r x^+)$</td><td><i>(fixpoint expr.)[†]</i></td></tr> <tr><td> try $f x^+$ catch x</td><td><i>(catch expr.)[†]</i></td></tr> <tr><td> throw x</td><td><i>(throw expr.)</i></td></tr> </table> <p>[†]where f is saturated</p>	let b in e	<i>(simple expr.)</i>	letS b in e	<i>(lazy let expr.)</i>	case s of $\{ a^+ \}$	<i>(case expr.)</i>	if c then e else e	<i>(if expr.)</i>	fix $(\lambda r. f r x^+)$	<i>(fixpoint expr.)[†]</i>	try $f x^+$ catch x	<i>(catch expr.)[†]</i>	throw x	<i>(throw expr.)</i>	<table style="border: none;"> <tr><td>$a ::= C x^* \rightarrow e$</td><td><i>(constructor alternative)</i></td></tr> <tr><td>$b ::= x = s$</td><td><i>(binding)</i></td></tr> <tr><td>$c ::= x \bowtie x$</td><td><i>(primitive comparison)</i></td></tr> </table> <p>simple expression:</p> <table style="border: none;"> <tr><td>$s ::= x$</td><td><i>(variable)</i></td></tr> <tr><td> n</td><td><i>(integer)</i></td></tr> <tr><td> $C x^*$</td><td><i>(constructor [application])</i></td></tr> <tr><td> $f x^*$</td><td><i>(function [application])</i></td></tr> <tr><td> $g x^*$</td><td><i>(global const. [application])</i></td></tr> <tr><td> $y x^+$</td><td><i>(variable application)</i></td></tr> <tr><td> $\otimes x^+$</td><td><i>(primitive operation)</i></td></tr> <tr><td> $\pi_n x$</td><td><i>(proj. of a product type)</i></td></tr> </table>	$a ::= C x^* \rightarrow e$	<i>(constructor alternative)</i>	$b ::= x = s$	<i>(binding)</i>	$c ::= x \bowtie x$	<i>(primitive comparison)</i>	$s ::= x$	<i>(variable)</i>	n	<i>(integer)</i>	$C x^*$	<i>(constructor [application])</i>	$f x^*$	<i>(function [application])</i>	$g x^*$	<i>(global const. [application])</i>	$y x^+$	<i>(variable application)</i>	$\otimes x^+$	<i>(primitive operation)</i>	$\pi_n x$	<i>(proj. of a product type)</i>
let b in e	<i>(simple expr.)</i>																																				
letS b in e	<i>(lazy let expr.)</i>																																				
case s of $\{ a^+ \}$	<i>(case expr.)</i>																																				
if c then e else e	<i>(if expr.)</i>																																				
fix $(\lambda r. f r x^+)$	<i>(fixpoint expr.)[†]</i>																																				
try $f x^+$ catch x	<i>(catch expr.)[†]</i>																																				
throw x	<i>(throw expr.)</i>																																				
$a ::= C x^* \rightarrow e$	<i>(constructor alternative)</i>																																				
$b ::= x = s$	<i>(binding)</i>																																				
$c ::= x \bowtie x$	<i>(primitive comparison)</i>																																				
$s ::= x$	<i>(variable)</i>																																				
n	<i>(integer)</i>																																				
$C x^*$	<i>(constructor [application])</i>																																				
$f x^*$	<i>(function [application])</i>																																				
$g x^*$	<i>(global const. [application])</i>																																				
$y x^+$	<i>(variable application)</i>																																				
$\otimes x^+$	<i>(primitive operation)</i>																																				
$\pi_n x$	<i>(proj. of a product type)</i>																																				

Fig. 1. Grammar of the simple core language

3.1 A Simple Functional Core Language

As an intermediate step in the compilation process, we want a language that is both simpler and more explicit, with special and primitive constructs, than External Core. The simple core language is (like GRIN) structured using supercombinators, which means that all lambdas need to be transformed to toplevel functions (lambda lifting). The grammar of the simple core language is given in Figure 1. Subexpressions and arguments are restricted to plain variables, achieved by introducing let expressions for complex subexpressions. An explicit distinction is made between top level functions and global constants (both have globally unique names, denoted by f and g). All constructors and primitive operations are fully saturated (all arguments applied). From the advanced type system in External Core, the types are simplified to the point where only the distinction between reference and primitive types remains. Strictness is explicit in this core language, using a strict let, a strict case scrutinee, and strict primitive variables. This core language has primitive (arithmetic) operations and an

if-construct for primitive comparison and branching. Exception handling is supported by a try/catch construct and a throw expression. Let expressions are not recursive; for recursive values a fixed point combinator is used instead. Selection from a product type could be done using a case expression, but is specialized with a projection expression.

3.2 Evaluation Model and Memory Layout

Almost all modern implementations of lazy functional languages use a variation of the G-machine, which introduced compiled graph reduction [6]. The evaluation model and memory layout described in this section are derived largely from a mix of the spineless tagless G-machine (STG), as used in GHC [12], and GRIN [3]. The machine model (from GRIN) consists of an environment and a heap. The environment maps variables to values, where values can be either *primitive values* or *heap references*. The environment is implemented as a stack of call frames. The heap maps references to *nodes*, where a node is a data structure consisting of a tag and zero-or-more arguments (values). Node tags specify the node’s type and contain additional metadata, in particular a bitmask denoting which of the node’s arguments are primitive values and which are references. This simplifies garbage collection. Using metadata in tags—as opposed to using info pointers, as in STG—makes handling tags more involved, but reduces the number of memory accesses.

Table 1. Node types

Node Type	Represents	Contents
C	Constructor	Constructor arguments
P	Partially applied function	Args. and number of missing args.
F	Fully applied function	All arguments for the function

Tags distinguish three basic node types (see Table 1). Both C- and P-nodes are in Weak Head Normal Form (WHNF), while F-nodes are not. Tags for C-nodes contain a unique number for the data type they belong to and the index of the constructor therein. The P- and F-tags contain a pointer to the applied function. The F-nodes are closures, that can be evaluated when required. To implement laziness, F-nodes are overwritten (updated) with the result after evaluation.

A program is a set of functions, each resulting in an evaluated node on the stack (i.e., a C- or P-node). Every function has a fixed list of arguments and consists of a sequence of instructions. Next, we introduce the instruction set, on the level of an assembly language (i.e., with variables as opposed to registers).

3.3 Assembly Language

Similar to GRIN, PilGRIM’s assembly language can be seen as a first-order, untyped, strict, and monadic language. The monad, in this case, is abstract and implicit. That is, the programmer can not access its internal state other than

through the predefined instructions. This internal state is, in fact, the state of the heap and the stack. Unlike GRIN, the syntax of PilGRIM’s assembly language is not based on a “built-in bind structure,” [3] but rather on Haskell’s do-notation. This, however, imposes considerably more restrictions, viz. in

$$pattern \leftarrow instruction ; rest$$

the variables in *pattern* are bound in *rest* to the corresponding parts of the result of *instruction*. Which pattern is allowed is determined by the instruction. The grammar of the assembly language (including extensions) is shown in Figure 2.

function implementation: $fun ::= f a^+ = block$ $g = block$ basic block: $block ::= instr;^* term$ instruction: $instr ::= x \leftarrow Store T a^*$ $x \leftarrow Push_{CAF} g$ $y \leftarrow PrimOp \otimes y y$ $y \leftarrow Constant n$ $T a^* \leftarrow Call call cont$ $x \leftarrow Force call cont$ terminator instruction: $term ::= Return T a^*$ $Jump call cont$ $Case call cont alt^+$ $If (y \bowtie y) block block$ $Throw x$	case alternative: $alt ::= T a^* \rightarrow block$ callable expression: $call ::= (Eval x)$ $(Eval_{CAF} g)$ $(TLF f a^+)$ $(Fix f a^*)$ evaluation continuation: $cont ::= ()$ $(Apply a^+)$ $(Select n)$ $(Catch x)$ node tag: $T ::= C_{con}$ F_{fun} P_{fun}^m argument or parameter: $a ::= x$ (<i>ref. var.</i>) y (<i>prim. var.</i>)
--	---

Fig. 2. Grammar of PilGRIM’s assembly language

Like the simple core language, PilGRIM programs are structured using supercombinators (i.e., only top-level functions have formal parameters). A program is a set of function definitions, where a function is defined by its name, its formal parameters, and a code block. A block can be thought of as a unit, in that control flow does not enter a block other than at its beginning and once a block is entered, all instructions in that block will be executed. Only the last instruction can redirect control flow to another block. Thus, we distinguish between *instructions* and *terminator instructions*. The former can not be the last of a block, whereas the latter must be. It follows, that a block is a (possibly empty) sequence of instructions, followed by a terminator instruction.

As discussed in Section 3.2, nodes consist of a tag and a list of arguments. When functions return their result, they do this in the form of a node on the top of the stack. This is why the pattern to bind variables to the result of a `Call` must be in the form of a tag with a list of parameters.

Basic instructions. We will continue with an informal description of all instructions that are essential for a lazy functional language. The appendix contains a more formal description of the semantics of the assembly language in an operational style.

First, we only consider top-level functions. A top-level function is called by forming a *callable*, using **TLF** with the name of the function to call and all the arguments to call it with. Given such a callable, **Call** pushes the return address and the arguments onto the stack and jumps to the called function's code block. In all cases, a **Call** has a corresponding **Return**. **Return** clears the stack down to (and including) the return address pushed by **Call**. Next, it pushes its result node (tag and arguments) onto the stack, before returning control flow to the return address.

Next, we consider calls to non-top-level functions. As discussed above, F-nodes contain fully applied functions. Therefore, F-nodes can be considered callable. On the level of PilGRIM's assembly language, however, different types of nodes can not be identified in the program. To this end, **Eval** takes a heap reference to any type of node, loads that node from the heap onto the stack, and turns it into a callable. Calling **Eval** x thus behaves differently for different types of nodes. Since C- and P-nodes are already in WHNF, a **Call** to such a node will implicitly return immediately (i.e., it simply leaves the node on the stack). F-nodes are called using the same mechanism as top-level functions.

In Figure 2, **Call** has another argument, namely a *continuation*. In PilGRIM's assembly language, continuations are transformations of the result of a call, so they can be seen as part of the return. The empty continuation, $()$, leaves the result unchanged. **Select** n takes the n^{th} argument from the C-node residing at the top of the stack and (if that argument is a reference) loads the corresponding node from the heap onto the stack. Similarly, **Apply** works on the P-node at the top of the stack. Given a list of arguments, **Apply** appends its arguments to those of the P-node. If this saturates the P-node (i.e., if this reduces the number of missing arguments to nil), the resulting F-node is automatically called, as if using **Call** and **Eval**.

Instead of returning, another way to transfer control flow from the end of a block to the beginning of another is by using **Case**. **Case** can be understood as a **Call** combined with a switch statement, or **Call** as a **Case** with only one alternative. After the callable returns a result and the continuation is applied, the C-node at the top of the stack is matched against a number of cases. If a case matches, the arguments of the node are bound to the corresponding parameters of the case and control flow is transferred to the case's block. Note that in this transfer of control flow, the stack is unaltered (i.e., the top-most return address remains the return address for the next **Return**).

Finally, a node can be written to the heap by the **Store** instruction. Since the heap is garbage collected, the address where a node is stored is not in the program's control. **Store** takes its arguments from the stack, allocates a new heap node, and then pushes the reference to the new heap node onto the stack.

Extensions. Because some patterns of instructions are very common and, thus far, the instruction set is very small, a few extra instructions and combined instructions are added as optimizations.

First, support for primitive values and operations is added. As a load immediate instruction, `Constant` produces a primitive constant. Primitive constants can be fed to primitive operations by means of `PrimOp`. Control flow can be determined by comparison operations on primitive values in `If`. The implementation of arithmetic operations in PilGRIM is such, that it can be seen as an independent coprocessor. By letting the operation be a parameter of `PrimOp` and `If`, the instruction set is easily extended with more primitive operations.

Second, we add support for functions in Constant Applicative Form (CAF). These are functions without arguments (i.e., global constants). They are stored in a fixed position on the heap and require special treatment in case of garbage collection [11]. To this end, `PushCAF` generates a constant reference to a function in CAF. Furthermore, `EvalCAF` is used to make such a function callable. `EvalCAF` can be interpreted as a `PushCAF`, followed by an `Eval`.

Third, there are some useful optimizations with regards to control flow instructions. The common occurrence of tail recursion in lazy functional programming languages calls for a cheaper means than having to `Call` every recursion. The `Jump` instruction is a terminator instruction that redirects control to the code block of the recursive call, without instantiating a new call frame on the stack. Another combination with calls is having a `Call` immediately followed by a `Store`. This happens in the case of evaluating references in a strict context. `Force` is a `Call`, followed immediately by a `Store` of the call's result.

Finally, for exception handling, we add a terminator instruction `Throw` and a continuation `Catch`. The latter places a reference to an exception handler on the stack. The former unwinds the stack, down to the first exception handler and calls that handler with the thrown value as argument.

3.4 Translation of the Core Language to the Instruction Set

The translation from the core language (presented in Section 3.1) to PilGRIM's assembly language (Section 3.3) is defined by a four-level scheme. The entry-point of the translation is \mathcal{T} (Figure 3). This scheme translates (strict) top-level expressions of the core language. It is quite straightforward, except maybe for the translation of function applications. At this point, the distinction must be made between saturated an unsaturated function application. The notation $\alpha(f)$ refers to the arity of function f , whereas $|\vec{x}|$ denotes the number of arguments in the core language expression.

Subexpressions can be translated for lazy evaluation (by means of scheme \mathcal{V}), or strict evaluation (scheme \mathcal{S}). Note that some subexpressions are only in scheme \mathcal{S} , because they are strict by definition. The lowest level of the translation is scheme \mathcal{E} . This scheme determines the calling method for every expression (i.e., both the callable *and* the continuation).

toplevel expression translation:

$$\begin{aligned}
\mathcal{T} \llbracket C \vec{x} \rrbracket &= \text{Return } C_c \vec{x} \\
\mathcal{T} \llbracket f \vec{x} \rrbracket &= \begin{cases} \text{Jump } \mathcal{E} \llbracket f \vec{x} \rrbracket & \text{if } \alpha(f) \leq |\vec{x}| \\ \text{Return } P_f \vec{x} & \text{if } \alpha(f) > |\vec{x}| \end{cases} \\
\mathcal{T} \llbracket s \rrbracket &= \text{Jump } \mathcal{E} \llbracket s \rrbracket \quad (\text{other simple expressions}) \\
\mathcal{T} \llbracket \text{let } x = s \text{ in } e \rrbracket &= x \leftarrow \mathcal{V} \llbracket s \rrbracket; \mathcal{T} \llbracket e \rrbracket \\
\mathcal{T} \llbracket \text{letS } x = s \text{ in } e \rrbracket &= x \leftarrow \mathcal{S} \llbracket s \rrbracket; \mathcal{T} \llbracket e \rrbracket \\
\mathcal{T} \llbracket \text{fix } (\lambda r. f r \vec{x}) \rrbracket &= \text{Jump } (\text{Fix } f \vec{x}) () \\
\mathcal{T} \llbracket \text{try } f \vec{x} \text{ catch } h \rrbracket &= \text{Jump } (\text{TLF } f \vec{x}) (\text{Catch } h) \\
\mathcal{T} \llbracket \text{throw } x \rrbracket &= \text{Throw } x \\
\mathcal{T} \llbracket \text{case } s \text{ of } \{ C \vec{y} \rightarrow e \} \rrbracket &= C \vec{y} \leftarrow \text{Call } \mathcal{E} \llbracket s \rrbracket; \mathcal{T} \llbracket e \rrbracket \\
\mathcal{T} \llbracket \text{case } s \text{ of } \{ \vec{a} \} \rrbracket &= \text{Case } \mathcal{E} \llbracket s \rrbracket [C \vec{y} \rightarrow \mathcal{T} \llbracket e \rrbracket \mid (C \vec{y} \rightarrow e) \leftarrow \vec{a}] \\
\mathcal{T} \llbracket \text{if } c \text{ then } p \text{ else } q \rrbracket &= \text{If } c \mathcal{T} \llbracket p \rrbracket \mathcal{T} \llbracket q \rrbracket
\end{aligned}$$

lazy subexpression translation:

$$\begin{aligned}
\mathcal{V} \llbracket g \vec{x} \rrbracket &= \begin{cases} \text{Push}_{\text{CAF}} g & \text{if } |\vec{x}| = 0 \\ h \leftarrow \text{Push}_{\text{CAF}} g; & \text{if } |\vec{x}| > 0 \\ \text{Store } F_{\text{ap}} h \vec{x} & \end{cases} \\
\mathcal{V} \llbracket f \vec{x} \rrbracket &= \begin{cases} \text{Store } P_f \vec{x} & \text{if } \alpha(f) > |\vec{x}| \\ \text{Store } F_f \vec{x} & \text{if } \alpha(f) = |\vec{x}| \\ h \leftarrow \text{Store } F_f \vec{y}; & \text{where } \vec{y} = x_1, \dots, x_{\alpha(f)} \text{ if } \alpha(f) < |\vec{x}| \\ \text{Store } F_{\text{ap}} h \vec{z} & \text{where } \vec{z} = x_{\alpha(f)+1}, \dots, x_{|\vec{x}|} \end{cases} \\
\mathcal{V} \llbracket y \vec{x} \rrbracket &= \text{Store } F_{\text{ap}} y \vec{x} \\
\mathcal{V} \llbracket \pi_n x \rrbracket &= \text{Store } F_{\text{sel}_n} x
\end{aligned}$$

strict subexpression translation:

$$\begin{aligned}
\mathcal{S} \llbracket n \rrbracket &= \text{Constant } n \\
\mathcal{S} \llbracket \otimes \vec{x} \rrbracket &= \text{PrimOp } \otimes \vec{x} \\
\mathcal{S} \llbracket x \rrbracket &= \text{Force } \mathcal{E} \llbracket x \rrbracket \\
\mathcal{S} \llbracket C \vec{x} \rrbracket &= \text{Store } C_c \vec{x} \\
\mathcal{S} \llbracket y \vec{x} \rrbracket &= \text{Force } \mathcal{E} \llbracket y \vec{x} \rrbracket \\
\mathcal{S} \llbracket \pi_n x \rrbracket &= \text{Force } \mathcal{E} \llbracket \pi_n x \rrbracket \\
\mathcal{S} \llbracket f \vec{x} \rrbracket &= \begin{cases} \text{Force } \mathcal{E} \llbracket f \vec{x} \rrbracket & \text{if } \alpha(f) \leq |\vec{x}| \\ \text{Store } P_f^m \vec{x} & \text{where } m = \alpha(f) - |\vec{x}| \text{ if } \alpha(f) > |\vec{x}| \end{cases}
\end{aligned}$$

evaluation expression translation:

$$\begin{aligned}
\mathcal{E} \llbracket x \rrbracket &= (\text{Eval } x) () \\
\mathcal{E} \llbracket \pi_n x \rrbracket &= (\text{Eval } x) (\text{Select } n) \\
\mathcal{E} \llbracket g \vec{x} \rrbracket &= \begin{cases} (\text{Eval}_{\text{CAF}} g) () & \text{if } |\vec{x}| = 0 \\ (\text{Eval}_{\text{CAF}} g) (\text{Apply } \vec{x}) & \text{if } |\vec{x}| > 0 \end{cases} \\
\mathcal{E} \llbracket f \vec{x} \rrbracket &= \begin{cases} (\text{TLF } f \vec{x}) () & \text{if } \alpha(f) = |\vec{x}| \\ (\text{TLF } f \vec{y}) (\text{Apply } \vec{z}) & \text{where } \vec{y}, \vec{z} \text{ as in } \mathcal{V} \llbracket f \vec{x} \rrbracket \text{ above if } \alpha(f) < |\vec{x}| \end{cases} \\
\mathcal{E} \llbracket y \vec{x} \rrbracket &= (\text{Eval } y) (\text{Apply } \vec{x})
\end{aligned}$$

Fig. 3. Translation from simple core to PilGRIM's assembly language

4 The PilGRIM Architecture

In this section, we describe a simplified variant of the PilGRIM architecture, which is complete with regards to the instruction set defined in the previous section. This variant is not pipelined and does not include many hardware optimizations.

The Structure of the Architecture

Essential to the understanding of the structure of PilGRIM, is the partitioning of the memory components. The main memory element is the *heap*. The heap contains whole nodes (i.e., addressing and alignment are both based on the size and format of nodes which is discussed in more detail in Section 4.3). Loads from and stores to the heap are performed in a single step. That is, memory buses are also node-aligned. For the larger part, the heap consists of external DDR-memory. However, PilGRIM has a small allocation heap (see Section 4.2) to exploit the memory locality present in typical functional programs.

PilGRIM's assembly language (Section 3.3) is based on a stack-model. The *stack* supports (random access) reading and pushing everything that is required for the execution of an instruction in a single cycle. The stack contains nodes, values, and call frames. Call frames always contain a return address and may contain an update continuation and zero or more application continuations. As discussed in Section 4.1, the stack is not implemented as a monolithic component, but split up to make parallel access less costly in hardware.

At the logistic heart of the architecture sits a *crossbar*, which connects the stack, the heap, and an ALU (used for primitive operations). The crossbar can combine parts from different sources in parallel to build a whole node, or anything else that can be stored or pushed on the stack in a single step. PilGRIM's control comes from a *sequencer*, that calculates instruction addresses, decodes instructions, and controls all other components of the architecture. The sequencer reads instructions from a dedicated *code memory*.

4.1 Splitting the Stack

The stack contains many different kinds of values: nodes read from memory, function arguments, return addresses, update pointers, intermediate primitive values, and temporary references to stored nodes. Every instruction reads and/or writes multiple values from/to the stack. All these data movements make the stack and the attached crossbar a critical central point of the core. The mix of multi-word nodes, single-word values, and variable-sized groups of arguments makes it very hard to implement a stack as parallelized as required, without making it big and slow. A solution to this problem is to split the stack into multiple special-purpose stacks. The second reason to split up the stack is to make it possible to execute a complete instruction at once (by having a separate stack for each aspect of an instruction).

The *return/update stack* contains the return addresses, update references, and counters for the number of nodes and continuations that belong to a callframe.

The *continuation stack* contains arguments to be applied or other simple continuations to be executed, between a return instruction and the actual jump to the return address. The *node stack* only contains complete nodes (including their tag). Values in the topmost few entries of the node stack can be read directly. Every instruction can read from and pop off any combination of these top entries. Reading from, popping from and pushing onto the node stack can take place simultaneously. The *reference queue* contains the references to the most recently stored nodes. The *primitive queue* contains the most recently produced primitive values.

The return stack and continuation stack are simple stacks with only push/pop functionality. The queues are register file structures with multiple read ports, that contain the n most recently written values. When writing a new value into a queue, the oldest value in the queue is lost.

A very parallel stack, such as the node stack, requires many read and write ports in its hardware implementation, which make it slow and big. By storing the top of the stack in separate registers, the number of read ports for the rest of the stack can be reduced, because the top element is accessed most often. Pushing can be faster, because the pushed data can only go to a single place and the *top of stack register* can be placed close to the local heap memory. The other stacks also have a top of stack register for fast access (not shown in Figure 4). All stacks are implemented using register files for the top few entries and backed up by a local memory. Transfers between the stack registers and the stack memory are handled automatically in the background by the hardware.

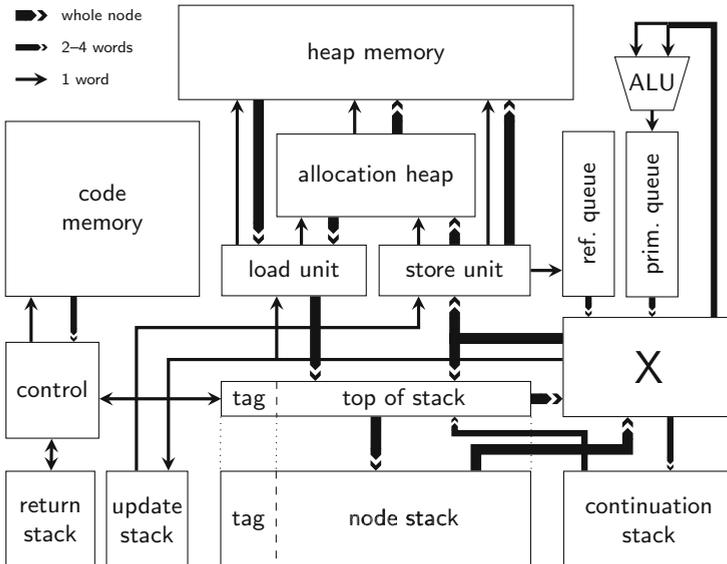


Fig. 4. The basic PilGRIM architecture

4.2 Allocation Heap

Typically as much as 25% of the executed instructions are stores, requiring a lot of allocation bandwidth. The percentage of instructions reading from the heap can be even higher than that. It is crucial that most loads and stores be fast, which can be achieved by using a small local memory.

We can make allocation more efficient, by storing all newly allocated nodes in a sizable buffer (the allocation heap) of several hundreds of nodes first. The allocation heap serves, in effect, as a fast, directly mapped, tagless data cache. We have chosen 512 elements of four words as the size for the allocation heap. This is 16 kilobytes of memory. The allocation heap exploits the temporal locality between the allocation of and reading back of the same data. To save bandwidth to external memory, nodes with a short lifetime are garbage collected while still in the allocation heap. This is implemented in hardware using a reference counting mechanism (extending the one-bit reference counters from SKIM [14]).

4.3 Hardware Component Sizes and Constraints

We have chosen the sizes of hardware components to be small enough to ensure that the cost in area and latency are reasonable, but also big enough to not limit performance too much. Some hardware constraints need transformations in the code generator to work around them.

- The datapath is 64 bits wide and all data in memory and registers is organized and addressed in 64 bit words.
- Nodes are limited to eight words (one tag word and seven values). The compiler has to transform the code to eliminate all constructors and function applications that are too big, as is done in the BWM and the Reduceron.
- The heap is divided in four word elements, because most of the nodes on the heap fit in four words [1,9]. Using wider elements adds a lot to hardware cost and only gives a small performance benefit. If a node is wider than four words, reading or writing nodes takes extra cycles. Updating a small F-node with a big result node is done using indirection nodes, as in the STG [12].
- Only the top four entries of the node stack can be read. The code generator ensures that all required values from the stack are in the top four nodes.
- Both the reference queue and the primitive queue are limited to the last 16 values produced. Older values need to be pushed onto the node stack.
- The continuation stack is two words wide. This is enough for most function applications [7]. For bigger applications, arguments can be pushed onto the continuation stack in multiple steps.

4.4 Instruction Set

The assembly language (defined in Section 3.3) and the instruction set differ only in two important details. First, all variables are replaced by explicit indices to elements on the stacks or queues. For reference values, copying/destructive reads

are explicitly encoded, to have accurate live reference information for garbage collection. Second, the structure of programs is transformed to a linear sequence of instructions, by using jump offsets for the else branches and case alternatives.

Instruction Set Encoding. All instructions can be encoded in 128 bits, although the majority of instructions fit within 64 bits. The mix of two instruction lengths is a trade off between fast (fixed length) instruction decoding and small code size (where variable length instructions are better). Small instructions are all arithmetic operations, constant generation, if-with-compare, and many other instructions with only a few operands. Operands are not only stack and queue indices, but can also be small constant values. The distinction between primitive and reference values is explicit in the operands. All instructions have a common header structure, with the opcode and a bitmask to optionally clear the queues or any of the top stack nodes.

Generating Instructions. The process of generating actual instructions from the assembly language is straightforward. Every constructor is assigned a unique number, where the numbers are grouped in such a way, that the lowest bits distinguish between the alternatives within a data type. Values on the node stack are indexed from the top by a number starting with zero, and values in the queues are indexed from the most recently produced entry in the queue. Every instruction that produces a result pushes its result on either the node stack or one of the queues. Thus the *sequence of executed instructions* within a supercombinator determines the translation of variables in the assembly to stack or queue indices (without any ‘register’ allocation). The last step is linking together all supercombinators, which assigns an instruction address to every use of a function name.

4.5 Hardware Implementation Strategy

The implementation of the PilGRIM is done in a series of Haskell programs, each one more detailed and lower level than the previous one. A high-level evaluator of the instruction set is the first step in this process. Following steps include: choosing sizes for memory structures, adding lower level optimizations, and splitting up functions to follow the structure of the hardware architecture.

We intend to use CλaSH [2] in the final step to produce a synthesizable implementation of the PilGRIM. CλaSH is a subset of Haskell that can be compiled to VHDL (a hardware description language supported by many existing tools).

We have gone through several iterations of the first few steps of this design process, starting from roughly GRIN as the instruction set on a simple single stack architecture, up to the current design. Making the step from a single cycle model to a fully pipelined architecture is ongoing work.

5 Evaluation and Comparison

For measurements in this section, we make use of the benchmark set of the Reduceron [9]. We selected these benchmarks, because they require only minimal

Table 2. Measurements of the performance of the instruction set

program	PilGRIM instrs. [million]	instr. rate to match PC [MHz]	arithmic instrs. [%]	node store instrs. [%]	Reduceron cycles [million]
PermSort	166	231	4	34	154
Queens	84	645	42	10	96
Braun	83	347	2	35	66
OrdList	98	271	0	32	94
Queens2	111	278	0	32	120
MSS	86	728	24	1	66
Adjoxo	23	378	23	13	37
Taut	39	352	5	18	54
CountDown	16	324	25	17	18
Clausify	46	284	5	17	67
While	46	359	9	24	56
SumPuz	277	338	12	18	362
Cichelli	29	587	19	6	39
Mate	343	528	15	6	628
KnuthBendix	15	302	8	18	16

support of primitive operations and the memory usage and run times are small enough to use them in simulation. The results come from an instruction-level simulator we developed for the PilGRIM. All measurements exclude the garbage collection overhead. GHC 6.10.4 (with the option `-O2`) was used for running the benchmarks on a PC and to produce to the simple core language for the PilGRIM code generator.

For 11 out of the 15 benchmarks in Table 2, 400 million PilGRIM instructions per second is enough to match the performance of GHC on a 2.66GHz Intel i5 PC. The benchmarks with a high percentage of arithmetic instructions do not perform so well on the PilGRIM, while the best performing ones are store intensive. The arithmetic performance of the PilGRIM could be improved, but the architecture has no inherent advantage on this aspect over other architectures.

For a deeply pipelined processor the performance in instructions says little about the actual performance. Pipeline stalls due to control flow and latency of memory operations are a big factor.

It is too early to compare with the Reduceron directly on performance, but we can compare the instruction set approach of the PilGRIM versus template instantiation in the Reduceron. For most benchmark programs the difference is fairly small (note that one is measured in instructions and the other in cycles). The Reduceron does better on Braun and MSS, due to requiring one step less in a critical small inner 'loop' in both programs. Where the numbers for PilGRIM are significantly lower, such as in Adjoxo and Mate, we believe they can entirely attributed to the optimizations applied by GHC.

5.1 Related Work

Between roughly 1975 and 1990, a lot of work was done on the design of Lisp machines and combinator based processors [14,13]. Big differences in implementation strategy and hardware technology leaves little to directly compare this work to. The only processor designs comparable to our work are Augustsson's Big Word Machine (BWM) [1] from 1991 and the recent work by Naylor and Runciman on the Reduceron [8,9]. Our work was inspired by the Reduceron, because of its promising results. The unusual architecture of the Reduceron leaves room (from a hardware perspective) for improvements and many alternative design choices. The BWM, Reduceron and PilGRIM have in common that they all focus on exploiting the potential parallelism in data movements inherent to functional languages, by reading multiple values in parallel from the stack and rearranging them through a large crossbar in every cycle. Advances in hardware technology allowed the Reduceron and the PilGRIM to go a step further than the BWM, by also using a wide heap memory and adding special-purpose stacks, that can be used in parallel. Both the BWM and the Reduceron choose to encode data constructors and case expression in functions for hardware simplicity. The Reduceron adds special hardware to speed up handling these function-application-encoded case expressions. The Reduceron is based on template instantiation, while the BWM uses a small instruction set, based on the G-machine [6]. Unfortunately, the BWM was only simulated and never built. The Reduceron has been implemented on an FPGA, achieving a clock speed close to 100MHz. The Reduceron executes one complete reduction step per cycle.

While the Reduceron achieves surprisingly high performance given its simplicity, the single cycle nature of its design is the limiting factor in performance. It will have a relatively low clock speed even in a silicon implementation, because the latency of memory reads and writes performed within every cycle. With the extensive use of pipelining the PilGRIM targets a high clock frequency which comes with a strong increase in the complexity of the design.

6 Conclusions

It is feasible to design an efficient processor for lazy functional languages using a high-level instruction set. We designed the instruction set to match closely with the source language, so that code generation is (relatively) simple. Lazy functional languages can expose a lot of low-level parallelism, even in an instruction set based architecture. Most of this parallelism is in data movements, and can be exploited by using wide memories and many parallel stacks. A large part of the abstraction overhead typically incurred by high-level languages can be removed, by adding specialized hardware.

Measurements have shown that the designed instruction set is better suited for the functional language domain than the instruction set of typical general purpose processors. Combined with the results from the Reduceron, we conclude that the potential exploitation of low-level parallelism is inherent to lazy

functional languages. This form of parallelism is not restricted to a single implementation strategy, but works both for template instantiation and a GRIN-derived instruction set. The advantage of using an instruction set based and pipelined design is that many existing hardware optimization techniques can be applied to the PilGRIM.

As for absolute performance numbers compared to desktop processors: the big open question is how often the pipeline of this core will stall on cache misses and branches. To become competitive with current general purpose processors, the PilGRIM needs to execute about 500 million instructions per second. We might be able to achieve that with the not unreasonable numbers of a 1 GHz operating frequency and executing an instruction every other cycle on average. The optimal trade off between frequency and pipeline stalls is an open question, and we might need to settle for a lower frequency in a first version, due to the complexity of a deep pipelined design.

Future Work. The next step is finishing a fully pipelined, cycle accurate simulation model with all optimizations applied. Most optimizations under consideration are to avoid pipeline stalls and to reduce the latency and bandwidth of memory accesses. Another area for optimization is the exploitation of potential parallelism in arithmetic operations. Then, we plan to transform the simulation model into a synthesizable hardware description of this architecture, and make it work on an FPGA, so that real programs can be benchmarked with it. Once this processor is complete, a lot of interesting research possibilities open up, like building a multicore system with it and making the core run threads concurrently by fine-grained multithreading.

Acknowledgements. We are grateful to the Reduceron team for making their benchmark set available online and providing raw numbers of the Reduceron. We also thank the anonymous reviewers, Kenneth Rovers, Christiaan Baaij and Raphael Poss for their extensive and helpful comments on earlier versions of this paper.

References

1. Augustsson, L.: BWM: A concrete machine for graph reduction. *Functional Programming*, 36–50 (1991)
2. Baaij, C.P.R., Kooijman, M., Kuper, J., Boeijink, W.A., Gerards, M.E.T.: *CLash*: Structural descriptions of synchronous hardware using haskell. In: *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, pp. 714–721 (September 2010)
3. Boquist, U.: *Code Optimisation Techniques for Lazy Functional Languages*. Ph.D. thesis, Chalmers University of Technology (April 1999), <http://www.cs.chalmers.se/~boquist/phd/phd.ps.gz>
4. Boquist, U., Johnsson, T.: The GRIN project: A highly optimising back end for lazy functional languages. *Implementation of Functional Languages*, 58–84 (1996)
5. Hennessy, J.L., Patterson, D.A.: *Computer Architecture: A Quantitative Approach*, 4th edn. Morgan Kaufmann, San Francisco (2006)

6. Johnsson, T.: Efficient compilation of lazy evaluation. In: SIGPLAN Symposium on Compiler Construction, pp. 58–69 (1984)
7. Marlow, S., Peyton Jones, S.L.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. In: ICFP, pp. 4–15 (2004)
8. Naylor, M., Runciman, C.: The reduceron: Widening the von neumann bottleneck for graph reduction using an FPGA. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 129–146. Springer, Heidelberg (2008)
9. Naylor, M., Runciman, C.: The reduceron reconfigured. In: ICFP (2010)
10. Nethercote, N., Mycroft, A.: The cache behaviour of large lazy functional programs on stock hardware. In: MSP/ISMM, pp. 44–55 (2002)
11. Peyton Jones, S.L.: The Implementation of Functional Programming Languages. Prentice-Hall, Inc., Englewood Cliffs (1987)
12. Peyton Jones, S.L.: Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *J. Funct. Program.* 2(2), 127–202 (1992)
13. Scheevel, M.: Norma: A graph reduction processor. In: LISP and Functional Programming, pp. 212–219 (1986)
14. Stoye, W.R., Clarke, T.J.W., Norman, A.C.: Some practical methods for rapid combinator reduction. In: LISP and Functional Programming, pp. 159–166 (1984)
15. Tolmach, A.: An external representation for the GHC core language (2001)

A Semantics

We chose to present the semantics on the level of assembly language, instead of the instruction set, in order to describe how the PilGRIM works without getting lost in less important details.

The processor state $\langle H|S|Q \rangle$ is a triple consisting of the *heap* H with bindings from references to nodes, the *stack* S (a sequence of stack frames), and a *queue* Q (a local sequence of temporary values). Each frame $\langle N|C|R \rangle$ on stack S has three parts: the local node stack N (a nonempty sequence of nodes), the optional local update/apply/select continuation stack C , and a return target R . Possible return targets are: returning a node or reference ($\mathbf{R}^{\mathbf{n}\mathbf{to}}/\mathbf{R}^{\mathbf{r}\mathbf{to}}$) to an instruction sequence, returning into case statement $\mathbf{R}\mathbf{case}$, continuing with the next stack frame $\mathbf{R}\mathbf{next}$, or completing the main function $\mathbf{R}\mathbf{main}$.

The semantics are written in an operational style, structured by the various modes of operation. These modes \mathcal{I} , \mathcal{C} , \mathcal{R} , \mathcal{E} , and \mathcal{W} , are explained below. Figure 5 contains the semantics for executing instructions indicated by the mode \mathcal{I} , with the instruction block as first argument. The callee mode \mathcal{C} is only a notational construct to avoid the combinatorial explosion between what is called and how it is called. Underlining of variables denotes the conversion from stack/queue indices to values (by reading from the stack and/or queue).

A program starts in evaluation \mathcal{E} mode, with the following initial state: $\langle g \mapsto \mathbf{F}_g \mid \langle \mathbf{F}_{\text{entry}} \vec{a} \mid \epsilon \mid \mathbf{R}\mathbf{main} \rangle \mid \epsilon \rangle$. The heap is initialized with a vector of bindings from constant references to nullary function nodes corresponding to all CAFs in the program. Initially the stack contains a single frame with the entry function with its arguments $\mathbf{F}_{\text{entry}} \vec{a}$ on the node stack. The queue and the extra continuation stack start empty, denoted by ϵ .

$\mathcal{I}(\mathbf{Store} \ t \ \vec{x}); \vec{v}$	$\langle H S Q \rangle \rightsquigarrow \mathcal{I} \ \vec{v}$	$\langle H, y \mapsto t \ \underline{\vec{x}} S y, Q \rangle$, $y = \mathit{newRef}(H)$
$\mathcal{I}(\mathbf{Push}_{\text{CAF}} \ g); \vec{v}$	$\langle H S Q \rangle \rightsquigarrow \mathcal{I} \ \vec{v}$	$\langle H S g, Q \rangle$
$\mathcal{I}(\mathbf{PrimOp} \ \otimes \ \vec{x}); \vec{v}$	$\langle H S Q \rangle \rightsquigarrow \mathcal{I} \ \vec{v}$	$\langle H S y, Q \rangle$, $y = \otimes \ \underline{\vec{x}}$
$\mathcal{I}(\mathbf{Constant} \ n); \vec{v}$	$\langle H S Q \rangle \rightsquigarrow \mathcal{I} \ \vec{v}$	$\langle H S n, Q \rangle$
$\mathcal{I}(\mathbf{Call} \ c \ e); \vec{v}$	$\langle H S Q \rangle \rightsquigarrow \mathcal{C} \ c \ e \ (\mathbf{R}^{\mathbf{n}\mathbf{to}} \ \vec{v})$	$\langle H S Q \rangle$
$\mathcal{I}(\mathbf{Force} \ c \ e); \vec{v}$	$\langle H S Q \rangle \rightsquigarrow \mathcal{C} \ c \ e \ (\mathbf{R}^{\mathbf{r}\mathbf{to}} \ \vec{v})$	$\langle H S Q \rangle$
$\mathcal{I}(\mathbf{Jump} \ c \ e)$	$\langle H S Q \rangle \rightsquigarrow \mathcal{C} \ c \ e \ \mathbf{R}\mathbf{next}$	$\langle H S Q \rangle$
$\mathcal{I}(\mathbf{Case} \ c \ e \ \vec{a})$	$\langle H S Q \rangle \rightsquigarrow \mathcal{C} \ c \ e \ (\mathbf{R}\mathbf{case} \ \vec{a})$	$\langle H S Q \rangle$
$\mathcal{I}(\mathbf{Return} \ t \ \vec{x})$	$\langle H S Q \rangle \rightsquigarrow \mathcal{R} \ (t \ \underline{\vec{x}})$	$\langle H S Q \rangle$
$\mathcal{I}(\mathbf{If} \ (x \bowtie y) \ t \ e)$	$\langle H S Q \rangle \rightsquigarrow \mathcal{I} \ t$	$\langle H S Q \rangle$, <i>if</i> $\underline{x} \bowtie \underline{y}$
	$\rightsquigarrow \mathcal{I} \ e$	$\langle H S Q \rangle$, <i>otherwise</i>
$\mathcal{I}(\mathbf{Throw} \ x)$	$\langle H S Q \rangle \rightsquigarrow \mathcal{W} \ \underline{x}$	$\langle H S \epsilon \rangle$
$\mathcal{C}(\mathbf{Eval} \ x) \ e \ r$	$\langle H S Q \rangle \rightsquigarrow \mathcal{E} \ \langle H \langle n \underline{e} r \rangle, S \underline{x} \rangle$, $n = \mathit{load}(H, \underline{x})$
$\mathcal{C}(\mathbf{Eval}_{\text{CAF}} \ g) \ e \ r$	$\langle H S Q \rangle \rightsquigarrow \mathcal{E} \ \langle H \langle n \underline{e} r \rangle, S g \rangle$, $n = \mathit{load}(H, g)$
$\mathcal{C}(\mathbf{TLF} \ f \ \vec{x}) \ e \ r$	$\langle H S Q \rangle \rightsquigarrow \mathcal{I} \ \vec{v} \ \langle H \langle \mathbf{F}_f \ \underline{\vec{x}} \underline{e} r \rangle, S Q \rangle$, $\vec{v} = \mathit{code}(f)$
$\mathcal{C}(\mathbf{Fix} \ f \ \vec{x}) \ e \ r$	$\langle H S Q \rangle \rightsquigarrow \mathcal{I} \ \vec{v} \ \langle H \langle \mathbf{F}_f \ \underline{\vec{x}} \mathbf{Update} \ y, \underline{e} r \rangle, S y, Q \rangle$, $y = \mathit{newRef}(H)$

Fig. 5. Semantics of the assembly language instructions

$\mathcal{R} \ n \langle H \langle N \mid C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{E} \langle H \langle n \mid C \mid R \rangle, S \mid \epsilon \rangle$
$\mathcal{E} \langle H \langle \mathbf{F}_f \bar{x} \mid C \mid R \rangle, S \mid u \rangle$	$\rightsquigarrow \mathcal{I} \ \bar{v} \langle H \langle \mathbf{F}_f \bar{x} \mid \mathbf{Update} \ u, C \mid R \rangle, S \mid \epsilon \rangle$, $\bar{v} = \text{code}(f)$
$\mathcal{E} \langle H \langle \mathbf{F}_f \bar{x} \mid C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{I} \ \bar{v} \langle H \langle \mathbf{F}_f \bar{x} \mid C \mid R \rangle, S \mid \epsilon \rangle$, $\bar{v} = \text{code}(f)$
$\mathcal{E} \langle H \langle n \mid \mathbf{Update} \ u, C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{E} \langle H, u \mapsto n \langle n \mid C \mid R \rangle, S \mid \epsilon \rangle$
$\mathcal{E} \langle H \langle n \mid \mathbf{Catch} \ h, C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{E} \langle H \langle n \mid C \mid R \rangle, S \mid \epsilon \rangle$
$\mathcal{E} \langle H \langle \mathbf{C}_c \bar{x} \mid \mathbf{Select} \ i, C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{E} \langle H \langle n \mid C \mid R \rangle, S \mid x_i \rangle$, $n = \text{load}(H, x_i)$
$\mathcal{E} \langle H \langle \mathbf{P}_f^m \bar{x} \mid \mathbf{Apply} \ \bar{a}, C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{E} \langle H \langle \mathbf{P}_f^{m- \bar{a} } [\bar{x}, \bar{a}] \mid C \mid R \rangle, S \mid \epsilon \rangle$, if $m > \bar{a} $
	$\rightsquigarrow \mathcal{E} \langle H \langle \mathbf{F}_f [\bar{x}, \bar{a}] \mid C \mid R \rangle, S \mid \epsilon \rangle$, if $m = \bar{a} $
	$\rightsquigarrow \mathcal{E} \langle H \langle \mathbf{F}_f [\bar{x}, a_0, \dots, a_{m-1}] \mid \mathbf{Apply} \ [a_m, \dots, a_{ \bar{a}-1}] , C \mid R \rangle, S \mid \epsilon \rangle$, if $m < \bar{a} $
$\mathcal{E} \langle H \langle n \mid \epsilon \mid \mathbf{Rnext} \rangle, \langle N \mid C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{E} \langle H \langle n \mid C \mid R \rangle, S \mid \epsilon \rangle$
$\mathcal{E} \langle H \langle n \mid \epsilon \mid \mathbf{R}^n \mathbf{to} \ \bar{v} \rangle, \langle N \mid C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{I} \ \bar{v} \langle H \langle n, N \mid C \mid R \rangle, S \mid \epsilon \rangle$
$\mathcal{E} \langle H \langle n \mid \epsilon \mid \mathbf{R}^r \mathbf{to} \ \bar{v} \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{I} \ \bar{v} \langle H, y \mapsto n \mid S \mid y \rangle$, $y = \text{newRef}(H)$
$\mathcal{E} \langle H \langle \mathbf{C}_c \bar{x} \mid \epsilon \mid \mathbf{Rcase} \ \bar{a} \rangle, \langle N \mid C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{I} \ \bar{v} \langle H \langle \mathbf{C}_c \bar{x}, N \mid C \mid R \rangle, S \mid \epsilon \rangle$, $\bar{v} = \text{selectAlt}(\bar{a}, c)$
$\mathcal{E} \langle H \langle n \mid \epsilon \mid \mathbf{Rmain} \rangle \mid \epsilon \rangle$	$\rightsquigarrow \langle H \langle n \mid \epsilon \mid \epsilon \rangle \mid \epsilon \rangle$ (end of program)
$\mathcal{W} \ x \langle H \langle N \mid \mathbf{Catch} \ h, C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{E} \langle H \langle n \mid \mathbf{Apply} \ x, C \mid R \rangle, S \mid h \rangle$, $n = \text{load}(H, h)$
$\mathcal{W} \ x \langle H \langle N \mid c, C \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{W} \ x \langle H \langle N \mid C \mid R \rangle, S \mid \epsilon \rangle$
$\mathcal{W} \ x \langle H \langle N \mid \epsilon \mid \mathbf{Rmain} \rangle \mid \epsilon \rangle$	$\rightsquigarrow \langle H \mid \epsilon \mid \epsilon \rangle$ (exit with exception)
$\mathcal{W} \ x \langle H \langle N \mid \epsilon \mid R \rangle, S \mid \epsilon \rangle$	$\rightsquigarrow \mathcal{W} \ x \langle H \mid S \mid \epsilon \rangle$

Fig. 6. Semantics of the stack evaluation/unwinding

Figure 6 contains the stack evaluation mode \mathcal{E} , and exception unwinding mode \mathcal{W} . Here, the order of the rules does matter: if multiple rules could match, the topmost one is taken. To avoid too many combinations, the rules for the evaluation mode are split up in multiple small steps. Returning a node in mode \mathcal{R} is (after pushing the node onto the stack) identical to evaluation (of a node loaded from the heap). Thus it can share the following steps.