Exceptional GRIN

Christof Douma

Thesis code: INF/SCR-05-55

18th May 2006

Contents

Chapter 1

Introduction

This thesis describes the implementation of a back-end for EHC[2], the Essential Haskell Compiler. The back-end is based on the works of the Ph.D. thesis of Urban Boquist on Code Optimisation Techniques for Lazy Functional Languages [6].

This thesis extends the intermediate language GRIN (Graph Reduction Intermediate Notation) with support for exceptions and describes the implementation of the *heap points-to analysis*, a global analysis which drives the compilation process.

1.1 Motivation

The approach used to implement a GRIN compiler is based on whole program analysis and aggressive optimisation. Motivation for this approach is the difference in memory usage and execution speed between programs written in a lazy functional language, like Haskell [12], and their imperative counterparts. This is not a surprise: lazy functional languages are more abstract and "further away from the machine" than imperative languages which gives a compiler a hard time to transform programs into efficient machine code. An aggressive optimisation approach will hopefully result in a better performance of a lazy functional program.

Boquist's thesis describes a compilation strategy to compile GRIN to a RISC architecture. However, there is no implementation available anymore. Also, various extensions exists for Haskell which need support in GRIN, for example imprecise exceptions [15].

This master thesis extends GRIN with support for exceptions and implements a GRIN compiler based on the description of Boquist's thesis. The output language of our GRIN compiler is the portable assembly language C-- [1, 19].

1.2 Research contributions

The main contributions of this thesis are:

- A description of the implementation of a *heap points-to analysis* (HPT) used by a GRIN compiler to approximate the *control flow* of a program. This analysis makes it possible to eliminate calls to all compile time unknown functions.

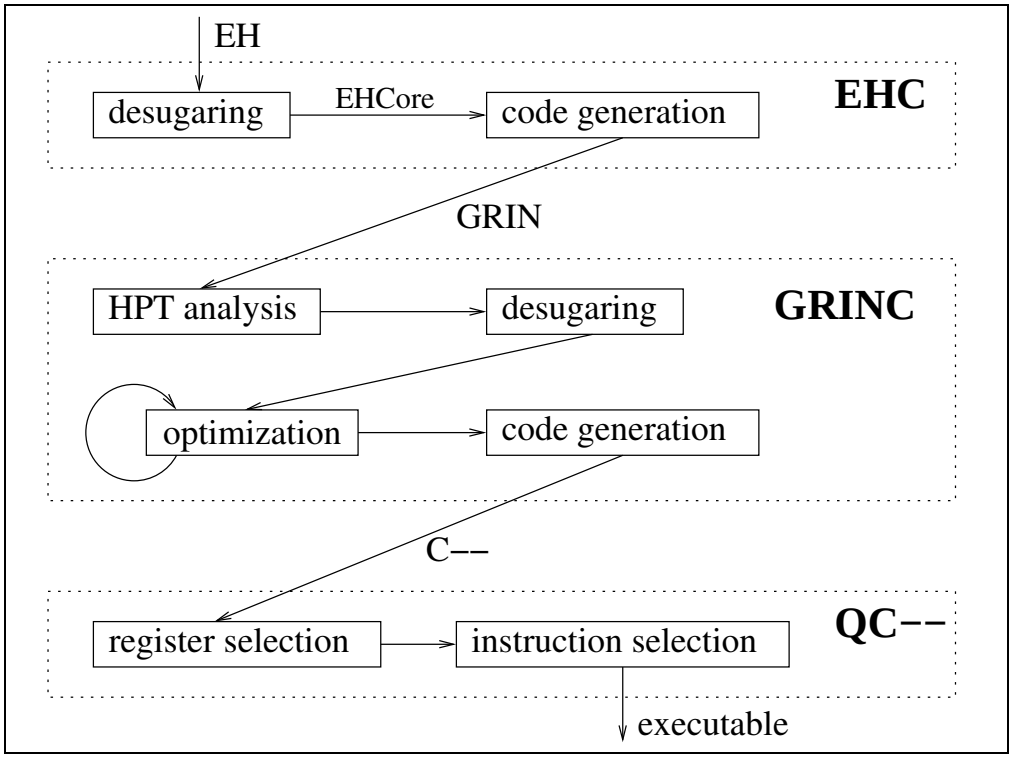

Figure 1.1: Overview of compilation process

- Exception support in the GRIN language. This makes efficient compilation of Haskell exceptions possible.
- Implementation of a back-end for EHC. Based on the Ph.D. thesis of Boquist and the findings on exceptions in this thesis, a GRIN compiler is implemented which compiles a program into C-- assembly.

1.3 Compiler overview

An overview of the compilation process is shown in Figure 1.1. It show the data flow between the various parts of the compilation process.

The input language of EHC is EH, a language in the spirit of Haskell. EH is translated by EHC into GRIN. This translation is briefly described in Section 2.3. The rest of this thesis discusses the syntax and semantics of GRIN and C--, and describes the various parts of GRINC, the GRIN compiler.

Chapter 2

GRIN

GRIN (Graph Reduction Intermediate Notation) is a language used to represent lazy functional programs in the back-end of a compiler. At this phase, the main task of a compiler is to bring a program from a high-level representation to an assembly representation. To assist the translation of a lazy functional program into assembly, GRIN contains both high-level and low-level constructs.

The high-level constructs in GRIN support the translation of Haskell to GRIN by the front-end. For example, Haskell values and thunks map directly to GRIN values, which can be stored to and loaded from memory with a single statement, as well as matched upon. The low-level constructs are represented by only one or a few assembly instructions and can, for example, load a single field of a node into a register.

In this chapter we give an overview of GRIN and cover its syntax, its semantics, and we address the translation of Haskell to GRIN.

2.1 Overview

GRIN is based on the G-machine [10] and is, like the G-machine, designed to implement lazy functional programs. GRIN is a simple language which makes as much operations as possible explicitly visible in the language itself. For example, the G-machine and STG-machine [13] have an *eval* primitive. In GRIN this is not a primitive but implemented as a normal GRIN function and, thus, subject to all optimisations of the compiler.

GRIN mixes features from both imperative and functional languages. For example, variables in GRIN cannot be updated by a programmer (like in a functional language). But unlike in a functional language, GRIN functions are not first class citizens. The design of GRIN imposes the following properties on the notation and semantics of the GRIN language:

- operations are executed sequentially, via the use of a build-in monad;
- function calls are to statically known functions;
- loops are not built-in primitives but are implemented by recursive tail calls;
- variables are named uniquely and have a single assignment point.

The property of sequential execution allows the use of side-effects. The side-effects possible in GRIN are limited to the minimal side-effect required for updating shared computations and creating self referencing values.

The other properties simplify the implementation of program-wide analysis and optimisation. The uniqueness of variables and their single assignment point is similar to the static single assignment (SSA) form[8] found in many compilers of imperative languages.

GRIN has notion of a store, or heap. A value can be stored in the heap, loaded from the heap, or be overwritten with a new value. Primitive operations are not defined by GRIN, but a foreign function interface exists to invoke functions outside GRIN.

2.1.1 Values

Values in GRIN can be categorised as follows:

- basic values, which are *int*, *char* and *float*;
- pointers, which hold references to heap allocated objects;
- nodes, which consist of a *tag* followed by zero or more basic values and pointers. Nodes reside on the heap, are stored in local variables, and returned by functions;
- tags, which normally identify nodes can also occur as a value. However, this is only possible internally, after transformations have been applied to the program.

GRIN does not impose any interpretation on node values, although it is aware of different tag categories. The tag categories represent the different uses of nodes by the front-end. The different categories are: *C-tags*, *P-tags*, and *F-tags*.

A C-tag identifies nodes which represent Haskell data values. Haskell functions can be applied to fewer arguments than needed. These functions are represented in GRIN by nodes identified with a P-tag. The lazy nature of computations in Haskell causes function calls to be evaluated only when their result is needed. When their result is not directly needed a *thunk* is created instead, which represent the function call and its arguments. In GRIN thunks are nodes, identified with an F-tag.

2.2 Syntax

Figure 2.1 shows the syntax of GRIN. This is a modified copy from the PhD thesis of Boquist[6]. Differences between this version and the version given by Boquist are:

- a **ffi** statement which is used to implement primitive operations with or without side-effect or to call functions in other programming languages;
- global variables of which the use is explained in Section 2.3.1;
- the use of curly brackets and semicolons for easy parsing. And a header to define the application name.

A GRIN program consist of a set of global variables and functions with the function *main* as the entry point. A function has a fixed number of arguments and returns a

prog	::=	**module** *string* { *global*∗ } { *binding*+ }	
global	::=	*var* ← **store** *val*	global variable
binding	::=	*var var*∗ = { *exp* }	definition
exp	::=	*exp* ; *λlpat* → *exp*	sequencing
	|	**case** *var* **of** { *alt* (; *alt*)∗ }	case
	|	*var val*∗	application, function call
	|	**unit** *val*	return value
	|	**store** *val*	allocate new heap node
	|	**fetch** *var offset*?	load heap node
	|	**update** *var val*	overwrite heap node
	|	**ffi** *string val*∗	foreign function call or primitive
alt	::=	*cpat* → { *exp* }	case alternative
val	::=	(*tag sval*∗)	complete node (constant tag)
	|	(*var sval*∗)	complete node (node variable)
	|	*tag*	single tag
	|	()	empty value
	|	*sval*	simple value
sval	::=	*literal*	constant
	|	*var*	variable
lpat	::=	(*tag var*∗)	constant node pattern
	|	(*var var*∗)	variable node pattern
	|	*tag*	single tag
	|	()	empty value
	|	*var*	variable
cpat	::=	(*tag var*∗)	constant node pattern
	|	*tag*	single tag
tag	::=	`#` *var*	tag identifier

? means zero or one
∗ means zero or more
+ means one or more

Figure 2.1: Syntax for GRIN

single GRIN value. The body of a function consists of a sequence of statements. These are sequenced by the monadic bind operator (denoted by "; $\lambda p \rightarrow$"):

$$
\begin{aligned}
&\mathit{statement}_1; \lambda \mathit{pattern}_1 \rightarrow \\
&\mathit{statement}_2; \lambda \mathit{pattern}_2 \rightarrow \\
&\ldots
\end{aligned}
$$

The bind operator binds the result of $\mathit{statement}_1$ to the identifiers in $\mathit{pattern}_1$, and continues with $\mathit{statement}_2$ of which the result is bound to $\mathit{pattern}_2$, etc. For those who are more familiar with imperative languages the following syntax will be more appealing; it resembles closely the intended meaning of a GRIN program:

$$
\begin{aligned}
&\mathit{pattern}_1 \leftarrow \mathit{statement}_1; \\
&\mathit{pattern}_2 \leftarrow \mathit{statement}_2; \\
&\ldots
\end{aligned}
$$

Other GRIN operations are:

- **unit** statements, to construct values;
- **case** statements to destruct node values and change the control flow based on the tags of the nodes;
- **ffi** statements to call functions outside GRIN. This statement can have side-effects;
- **store** statements to store a node in the heap;
- **fetch** statements to load (a field of) a node from the heap;
- **update** statements to overwrite nodes on the heap.

A part of this syntax is not considered as valid input by the current implementation. It is used to transform GRIN into a simple form which makes code generation easier. These constructs are: single tags; tag variables; and the offset field of a **fetch** statement. The **update** statement is only expected in the **eval** function, which is introduced later.

Tag values start with a hash character (#) followed with the category character. I.e., a C-tag would start with #C. As a convention we name the C-tags after their data constructors and the P- and F-tags after the function the represent. In addition, we prepend the function name with the number of missing parameters in a P-tag. Example tag names are: #*CTrue* for the Haskell value *True*, #*P2foo* a partial application of the Haskell function *foo* missing two arguments, and #*Ffoo* for a thunk of the function *foo*.

The GRIN version introduced here misses a few constructs used in the GRIN output of EHC. These constructs are used by EHC to support extensible records. As we will not discuss extensible records in this thesis we will ignore them. Chapter 7.2, which discusses future work, will discuss these constructs briefly.

2.3 Compiling to GRIN

A Haskell front-end typically translates Haskell to some small core representation. This core representation is then optimised and translated into code for the back-end [2, 3, 14]. A front-end targeting GRIN does the following:

- Lift local function declarations to the top level. As GRIN can only express top-level functions, any local function declaration is lifted to the top-level by applying lambda lifting. Lambda lifting replaces each free variable[1] in a local function with a new function argument. The resulting function lacks free variables and can safely be moved to the top-level. Functions in this form are also known as combinators.
- Each combinator is translated into a GRIN function. The arguments of the combinator are, in GRIN, passed as pointers to nodes in memory. These nodes represent suspended computation (thunks) or values in weak head normal form (WHNF).

Haskell combinators have the property that they always return a value in WHNF. Translated to GRIN, this property means that a GRIN function always returns some C- or P-node by evaluating a thunk (F-node), calling another combinator, or constructing a C- or P-node with a unit statement. The resulting node returned by a GRIN function is the node itself, *not* a pointer to a node in memory.

While each GRIN function is passed pointers to nodes in memory, it is also possible to pass basic values as arguments of GRIN functions. Passing complete nodes however creates difficulties: a node consist of a variable number of values, which makes implementing a code generator harder and is not allowed by the current implementation. However, the heap points-to analysis described in Chapter 3 is able to find the maximal number of values needed to represent a node variable, which makes it possible to split the node argument into a tag variable and a fixed number of basic values and pointer arguments.

A small EH program and its GRIN counterpart are given in Figure 2.2. The program calculates the integer value 42 in such a way that it demonstrates most of the GRIN constructs. The top-level EH expression is bound to the GRIN function main.

Before inspecting the example, we first discuss the special role of **eval** and **apply** in the translation.

Role of eval and apply

Graph reduction systems force, or evaluate, a suspended computation whenever its value is needed. In GRIN this is done by the function **eval**. Contrary to other graph reduction systems [10, 13], **eval** is expressed as a GRIN function. The **eval** implementation for the above example is shown in Figure 2.3. The semantics of **eval** is as follows: it takes a pointer to a node as its argument, loads that node from memory, and checks its tag:

- if it is a C-tag: the node represents a Haskell value in WHNF and is returned;
- if it is a P-tag: the node represents a Haskell partial application and is returned;
- if it is a F-tag: the node represents a thunk. To obtain its value we call the function represented by the F-node and pass the arguments inside the node.

Finally, the node in memory is updated with the result of the function call. This way, the thunk is overwritten with its result and the next time it is evaluated the result of the thunk is directly available.

[1] variables referring to other top-level functions are not considered free in this process.

```
EH
    let data Tuple a b = Tuple a b
    in
    let snd = λ(Tuple _ b) → b
        mk  = λf x         → f x
    in snd (mk (Tuple 1) 42)
GRIN
    module "example"
    { main_caf ← store (#Fmain)
    }
    { Tuple x1 x2 = {unit (#CTuple x1 x2)}
    ; snd t       = {eval t; λtn →
                     case tn of
                       {(#CTuple a b) → {eval b}
                       }
                    }
    ; mk f x3     = {eval f; λfn →
                     apply fn x3
                    }
    ; main        = {store (#CInt 1);       λs1 →
                     store (#P1Tuple s1); λs2 →
                     store (#CInt 42);     λs3 →
                     store (#Fmk s2 s3);   λs4 →
                     snd s4
                    }
    }
```

Figure 2.2: Haskell to GRIN example

```
eval p
  = {fetch p; λn →
     case n of
       {(#CInt v)          → {unit n}
       ;(#CTuple aa bb)    → {unit n}
       ;(#P1Tuple a1)      → {unit n}
       ;(#P2Tuple)         → {unit n}
       ;(#Fsnd a4)         → {snd a4; λrsnd →
                              update p rsnd; λ() →
                              unit rsnd
                             }
       ;(#Fmk a5 a6)       → {mk a5 a6; λrmk →
                              update p rmk; λ() →
                              unit rmk
                             }
       }
     }
apply pn arg
  = {case pn of
       {(#P1Tuple a1) → { Tuple a1 arg}
       ;(#P2Tuple)    → {unit (#P1Tuple arg)}
       }
     }
```

Figure 2.3: Eval and apply functions for Haskell to GRIN example

The **apply** function, also shown in Figure 2.3, is used to implement partial applications of functions. These partial applications are represented in GRIN by P-nodes. Whenever an argument must be applied to a partial application, the P-node and a pointer to the "to be applied argument" are passed to the GRIN function **apply**. Depending on the number of missing arguments, **apply** does the following:

- if the P-tag represents a partial application lacking more than one argument, then a new P-node is build with the newly applied argument appended to it;
- if the P-tag represents a partial application lacking exactly one argument, then the function represented by the P-tag is called with the arguments stored in the P-node and the newly applied argument.

To apply a partial application to more than one argument at once, a sequence of **apply** calls could be used. But this would involve building P-nodes for each applied argument. Instead, GRIN allows to call **apply** with any non-zero number of arguments. P-nodes are then build in one step with all the available arguments.

The eval and apply functions can be defined as normal GRIN functions. But the semantics of these functions is used by the compiler to make the heap points-to analysis more efficient. In order to avoid unnecessary parsing and analysis, these functions are compactly represented by two special purpose maps, making the apply and eval

$$
\begin{array}{lll}
\mathit{evalmap} & = \{ & \#\mathit{CInt} \quad 1 \rightarrow \textbf{unit} \\
& ; & \#\mathit{CTuple} \quad 2 \rightarrow \textbf{unit} \\
& ; & \#\mathit{P1Tuple} \; 1 \rightarrow \textbf{unit} \\
& ; & \#\mathit{P2Tuple} \; 0 \rightarrow \textbf{unit} \\
& ; & \#\mathit{FTuple} \quad 2 \rightarrow \mathit{Tuple} \\
& ; & \#\mathit{Fsnd} \quad 1 \rightarrow \mathit{snd} \\
& ; & \#\mathit{Fmk} \quad 2 \rightarrow \mathit{mk} \\
& \} & \\
\mathit{applymap} & = \{ & \#\mathit{P1Tuple} \; 1 \rightarrow \mathit{Tuple} \\
& ; & \#\mathit{P2Tuple} \; 0 \rightarrow \#\mathit{P1Tuple} \\
& \} &
\end{array}
$$

Figure 2.4: Eval and apply maps for Haskell to GRIN example

functions redundant as far as a GRIN compiler is concerned.[2]

The maps are named the eval- and the applymap. The map representation of the eval and apply functions of the example are shown in Figure 2.4. Each entry in these maps corresponds with an alternative of the corresponding case statement. The key of an entry is a tag and the size of the node represented by that tag. In the evalmap, an F-tag maps to a function identifier, all other tags map to **unit**. The applymap maps P-tags to either a new P-tag or a function identifier.

Example

Let we continue with the example in Figure 2.2. The EH version shows *what* is calculated: first, a partial applied tuple with as first argument the integer 1, is applied to the integer 42 via the *mk* function. Secondly, the second field is selected from the constructed tuple. This field contains the value 42.

The GRIN version shows rather *how* the result is calculated. The GRIN program starts with storing three nodes on the heap. It stores a *Int* value 1, a partial application of *Tuple* with as first argument the pointer to the *Int* value, another *Int* with the value 42, and finally a *mk* thunk with the partial application and the second *Int* value as its arguments. The final statement is a call to the function *snd*, which calculates its result as follows:

1. *snd* evaluates its argument, which is a thunk representing the EH expression *mk* (*Tuple* 1) 42. This thunk is evaluated by calling the function *mk*:
   (a) *mk* evaluates its argument, which is the partial application *Tuple* 1, and misses one argument. This node is already in WHNF, thus this value is simply loaded from memory and stored in the variable *fn*.
   (b) the partial application is applied to the second argument. This results in a call to the *Tuple* function. The *Tuple* function returns a node representing a tuple of its arguments.

[2]Note that the naming convention used here makes the maps trivial. Future version of GRIN might do without these maps.

2. The result of evaluation is thus a tuple. This tuple is scrutinized in the case statement of *snd*, which matches the only available alternative. With the match, the first field of the tuple is bound to the variable a, the second to b.

3. The result of *snd* is the evaluation of b, the second argument of the tuple, which is the node (#*CInt* 42).

The result of *snd* is a node representing the integer 42. This is also the last statement of the *main* function, and is thus also the value returned by the program.

2.3.1 Global variables

One class of functions is treated special: top-level functions without arguments, also known as Constant Applicative Forms (CAF). A CAF has the nice property that its value is constant because it has no arguments. Thus, all CAF thunks have the same form: an F-node without fields. Because a CAF has no arguments, all calls to a CAF will result in the same value. Thus, a thunk of a CAF can be shared throughout the program. In order to do so, CAF thunks are allocated as global variables before the program starts executing. Any reference to the CAF thunk goes through that global variable.

Global variables are always pointers to nodes in the heap and cannot contain other values. Beside the use of global variables to encode thunks of CAFs, global variables can also be used to pre-build nodes. Nodes which are commonly used and only contain constant values or pointers to other global nodes can be pre-build and stored as global variables. This transformation is called common subexpression elimination. Global variables give the front-end, which performs the common subexpression elimination, a place to pre-build the common constant expressions.

2.4 Semantics

The operational semantics of GRIN given in this section is copied from the PhD thesis of Boquist [6]. Note that the differences in the syntax can also also be found in the schematic functions defined in this section.

2.4.1 Semantic framework

Figure 2.5 shows the semantic framework in which the GRIN semantics is explained. It describes the domains of the semantic functions and a set of utility functions which are used in the semantic functions.

The semantics is split in three parts. The value semantics ($\mathcal{V}$); the program semantics ($\mathcal{A}$); and the expression semantics ($\mathcal{E}$). The value semantics is not much more than the identity function.

The semantic functions use a *store* (σ), which maps heap locations to values; and a *environment* (ρ), which maps variables to values. The store, together with the utility function *newloc* encodes the heap.

Semantic functions:

$\mathcal{A} :: prog \rightarrow (Value, Store)$
$\mathcal{E} :: exp \rightarrow Env \rightarrow Store \rightarrow (Value, Store)$
$\mathcal{V} :: val \rightarrow Env \rightarrow Value$

Semantic domains:

$Value = i \mid t \mid (t\ v_1 \ldots v_n) \mid l \mid () \mid \perp$
$Env = Var \rightarrow Value$
$Store = Loc \rightarrow Value$

Semantic variables:

$i \in Int, t \in Tag, l \in Loc, v \in Value$

Syntactic variables:

$x \in var, e \in val, m \in exp, k \in exp, c \in tag$

Utility functions:

$sel_i\ (v)$	-	select the i:th component of the node value v
$selalt_t\ (alt_1 \ldots alt_n)$	-	select one of the alternatives matching the tag t, return the bound variables and the alternative body
newloc	-	return a new (unused) heap location
getfunction (*fun*)	-	find the global function definition *fun*, return its parameters and body
runexternal $(ext, x_1 \ldots x_n)$	-	execute the external operation *ext* with the arguments $x_1 \ldots x_n$

Figure 2.5: Semantic framework, domains and utilities

$$
\begin{array}{lll}
\mathcal{V}\,[(c\ v_1 \ldots v_n)]\ \rho = (c & \mathcal{V}\,[v_1]\ \rho \ldots \mathcal{V}\,[v_n]\ \rho) \\
\mathcal{V}\,[(x\ v_1 \ldots v_n)]\ \rho = (\mathcal{V}\,[x]\ \rho\ \mathcal{V}\,[v_1]\ \rho \ldots \mathcal{V}\,[v_n]\ \rho) \\
\mathcal{V}\,[x\ ] & \rho = \rho\,(x) \\
\mathcal{V}\,[i\ ] & \rho = i \\
\mathcal{V}\,[l\ ] & \rho = l \\
\mathcal{V}\,[()] & \rho = ()
\end{array}
$$

Figure 2.6: GRIN value semantics

$$
\begin{array}{l}
\mathcal{A} :: \mathit{prog} \rightarrow (\mathit{Value}, \mathit{Store}) \\
\mathcal{A}\,[\textbf{module}\ \texttt{"name"} \\
\quad \{x_1 \leftarrow \textbf{store}\ e_1; \ldots; x_n \leftarrow \textbf{store}\ e_n\} \\
\quad \{\ldots; \mathit{main} = m; \ldots\}] = \textbf{let}\ l_1 = \mathit{newloc} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad l_n = \mathit{newloc} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \rho = \{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \sigma = \{l_1 \mapsto e_1, \ldots, l_n \mapsto e_n\} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathcal{E}\,[m]\ \rho\ \sigma
\end{array}
$$

Figure 2.7: GRIN Program semantics

The semantic value domain includes the same values as the syntactic domain. It includes tags, nodes, pointers, basic values and the empty value. A special *undefined* value ($\perp$) is included. This value is used to signal "program failure" but can also occur in normal programs.

Not all values are meaningful in all contexts. Different meta variables are used to limit the possible values. A **fetch** operation, for example, needs a pointer (denoted by the meta variable l) as its argument. If this is not the case the semantics of the program is undefined.

2.4.2 Value semantics

The value semantics is defined by the function $\mathcal{V}$ in Figure 2.6. The only thing happening in this function is a lookup in the environment. The rest is simply a identity mapping from the syntactic domain to value domain.

2.4.3 Program semantics

The program semantics ($\mathcal{A}$) is described in Figure 2.7. This function defines the semantics over the whole program. It creates an initial store and environment from the global variables and passes them, along with the body of *main*, to the semantic function $\mathcal{E}$. The store contain the nodes defined by the global variable; the environment contains the names of the global variables. These variables are pointers to the nodes in the store. The result of the program is the result of the body of the *main* function.

2.4.4 Expression semantics

The function $\mathcal{E}$, as defined in Figure 2.8, describes the semantics of GRIN expressions. It is not a 'pure' function: the *runexternal* utility function can have side effects not visible in the semantic function. For example, it can perform read/write operations on files.

Variable binding

Binding of variables occurs in several places: in the patterns of the bind operator between the statements; in the pattern alternatives of a case statement; and the formals of a function definition and in the definition of the global variables. All these bindings, except those of global variables, are local to the function in which they occur.

bind operator

The binding which occurs in the patterns of the bind operator can have different forms. For a single variable the result of a statement is bound to that variable. When a pattern expresses a complete node each field of the result is bound to the corresponding variable in the pattern. For example:

$$\textbf{unit}\ (\#\mathit{CCons}\ 2\ b); \lambda(\#\mathit{CCons}\ h\ t) \rightarrow \ldots$$

The result of the **unit** statement is the node $(\#\mathit{CCons}\ 2\ b)$. The pattern following the **unit** statement results in the bindings h to 2 and t to b. The tag in the result must match the one in the pattern. If this is not the case the semantics is undefined.

Case statement

A case statement will select one of its alternatives based on the tag of the node to be scrutinized. One, and only one, alternative must match, or the program is undefined.

When the alternative is chosen the fields of the scrutinized node are bound to the variables of the matching pattern. This works in the same way as the bind operator. The execution continues with the body of the alternative in question.

Function calls and ffi

An important property of function calls is that a function call in GRIN is *always* to some globally (compile time) known function. When all functions are known, *inter-procedural register allocation* and aggressive inling can be performed which result in better code quality.

A function call binds the actuals to the formals of the called function, executes its body and continues with the statement following the call.

A **ffi** statement is like a function call but to a function outside GRIN. This can be a primitive (which always start with `prim`), or to some function defined in a foreign language (for example C).

$$
\begin{array}{lcl}
\mathcal{E}\,[m; \lambda p \to k]\,\rho\,\sigma & = & \textbf{let } (v, \sigma') = \mathcal{E}\,[m]\,\rho\,\sigma \\
 & & \quad \rho' = \rho\,[p \mapsto v] \\
 & & \textbf{in } \mathcal{E}\,[k]\,\rho'\,\sigma' \\
\mathcal{E}\,[\textbf{case } e \textbf{ of } alt_1 \ldots alt_n]\,\rho\,\sigma = & & \\
\quad \textbf{case } \mathcal{V}\,[e]\,\rho \textbf{ of} & & \\
\quad\quad (t\ v_1 \ldots v_n) \to & \textbf{let } (x_1 \ldots x_n, k) = selalt_t\ (alt_1 \ldots alt_n) & \\
 & \quad\quad \rho' = \rho\,[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] & \\
 & \textbf{in } \mathcal{E}\,[k]\,\rho'\,\sigma & \\
\quad\quad t \to & \textbf{let } (\_, k) = selalt_t\ (alt_1 \ldots alt_n) & \\
 & \textbf{in } \mathcal{E}\,[k]\,\rho\,\sigma & \\
\mathcal{E}\,[\textbf{unit } e]\,\rho\,\sigma & = & (\mathcal{V}\,[e]\,\rho, \sigma) \\
\mathcal{E}\,[\textbf{store } e]\,\rho\,\sigma & = & \textbf{let } l = \quad newloc\ (\sigma) \\
 & & \quad \sigma' = \sigma\,[l \mapsto \mathcal{V}\,[e]\,\rho] \\
 & & \textbf{in } (l, \sigma') \\
\mathcal{E}\,[\textbf{update } x\ e]\,\rho\,\sigma & = & \textbf{let } l = \quad \rho\,(x) \\
 & & \quad \sigma' = \sigma\,[l \mapsto \mathcal{V}\,[e]\,\rho] \\
 & & \textbf{in } ((), \sigma') \\
\mathcal{E}\,[\textbf{fetch } x]\,\rho\,\sigma & = & \textbf{let } l\ = \rho\,(x) \\
 & & \quad v = \sigma\,(l) \\
 & & \textbf{in } (v, \sigma) \\
\mathcal{E}\,[\textbf{fetch } x\ offset]\,\rho\,\sigma & = & \textbf{let } l\ = \rho\,(x) \\
 & & \quad v = \sigma\,(l) \\
 & & \textbf{in } (sel_{offset}\ (v), \sigma) \\
\mathcal{E}\,[\textbf{ffi } fun\ e_1 \ldots e_n]\,\rho\,\sigma & = & \textbf{let } v_1 \quad = \mathcal{V}\,[e_1]\,\rho \\
 & & \quad \ldots \\
 & & \quad v_n \quad = \mathcal{V}\,[e_n]\,\rho \\
 & & \textbf{in } (runexternal\ (fun, v_1 \ldots v_n), \sigma) \\
\mathcal{E}\,[fun\ e_1 \ldots e_n]\,\rho\,\sigma & = & \textbf{let } (x_1 \ldots x_n, k) = getfunction\ (fun) \\
 & & \quad v_1 \quad = \mathcal{V}\,[e_1]\,\rho \\
 & & \quad \ldots \\
 & & \quad v_n \quad = \mathcal{V}\,[e_n]\,\rho \\
 & & \quad \rho' \quad = \rho\,[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\
 & & \textbf{in } \mathcal{E}\,[k]\,\rho'\,\sigma
\end{array}
$$

Figure 2.8: GRIN expression semantics

Built-in monadic operations

There are several monadic operations: **unit**, **store**, **fetch** and **update**. The *unit* operation is used to construct nodes. When it is the last statement this node is returned by the function. Otherwise the constructed node is bound to the pattern which follows it.

The **store** operation allocates space on the heap and stores the node passed to it. The allocated location is returned.

The **fetch** operation does the reverse of **store**. It takes a location and returns the node which is stored in that location. If an *offset* is given, only that field will be fetched from memory and returned. With an *offset* of zero the tag of the node in memory is returned.

The **update** operation overwrites memory locations with new nodes. Its only effect is its side effect and it returns an empty value.

Chapter 3

Heap points-to analysis

The Heap Points-To analysis (abbreviated by HPT) is an important analysis performed on GRIN programs. It is used by many transformations, both for simplification and for optimisation (See Chapter 5).

HPT is an abstract interpretation of a program. The result of the HPT is a pessimistic approximation of the contents of the variables and the heap. Thus, the absence of a value in a variable or heap location in the approximation is proof of the absence of the value in that particular variable or heap location in a run of the program.

This chapter describes the implementation of HPT. We discuss the implementation in a top-down way: we first show the results before explaining how to get these results. But first we start by an overview of the design of HPT.

3.1 Overview and design

The result of HPT is a set of abstract values; one for each variable and one for each store statement. An abstract value is an estimation of the possible GRIN values bound to a variable or stored in a memory location. Abstract values can be a set of pointers to abstract memory locations, or a set of abstract nodes with as fields sets of pointers. An example of an abstract node is #*CTuple* $[\{1,2\},\{3,4\}]$. An abstract pointer value is $\{1,2\}$, which represents a pointer to the first or second store location. The abstract value $\{1,2\}$ tells us that the variable bound to this value will, in a run of the program, hold a pointer to a location returned by the first or second store statement; never any other location.

The values derived by HPT are used by optimisations, in particular for the elimination of unreachable case alternatives. A good example is the **eval** function. The **eval** function contains a case statement listing all tags used in the program. But at every call site the pointers passed to **eval** reference only to a few of all those tags. An important transformation is the specialisation of this **eval** function for each call site at which the function is inlined. The result of HPT is used to prove the absence of a tag for the current call site, which makes it safe to remove the corresponding alternative from the case statement of **eval**.

To estimate the heap contents HPT analyses the complete program, which makes it

hard to keep the runtime of the compiler within reasonable limits. The design of HPT shown in this chapter ignores any form of control flow. The estimates made by HPT are worse this way, but HPT is more scaleble and is guaranteed to terminate as described by Urban Boquist [7].

The design of HPT is as follows:

- Each store statement is represented by exactly one location in an abstract heap. This location hold all the possibly nodes stored by that statement.
- Actual arguments of function calls are joined and used as the value of the corresponding formal arguments. For example: the function *foo a b* = ... is called twice: *foo* x_1 y_1 and *foo* x_2 y_2. In HPT, the formal argument a will be bound to the values of the actuals x_1 and x_2. The formal argument b will be bound to the values of y_1 and y_2.
- Abstract nodes with the same tag but with different fields are joined field by field. As an example take two abstract nodes #*CTuple* $[\{1\},\{3\}]$ and #*CTuple* $[\{2\},\{4\}]$. These joined form the abstract node #*CTuple* $[\{1,2\},\{3,4\}]$.
- Any basic value is represented as one single value, $\{B\}$. An empty value is represented by $\bot$.

The HPT results are calulated in two steps: a set of equations is derived from a GRIN program which are then solved by a *fixpoint computation*. A fixpoint computation is a common technique to solve a set of (mutual dependent) equations; an excelent introduction is given in "The Priciples of Program Analysis" [9].

3.2 Analysis result

Figure 3.1 gives the domains of the abstract heap and environment. We keep the notation close to the GRIN syntax. However, HPT and GRIN differ in the notation of nodes. An abstract node, as used by HPT, is of the form #*Tag* $[$*field1*$,\ldots,$*fieldn*$]$ while the syntax of nodes in GRIN is (#*Tag field1* ... *fieldn*). The difference between the abstract node and the GRIN node lies in the domains used for their fields. The separation of notation is necessary for the explanation of the transformations (Chapter 5) which uses GRIN syntax and HPT results together.

The HPT result of the GRIN program in Figure 2.2 on page 12 is represented in Figure 3.2. We suggest to keep both figures at hand while reading the rest of this section.

Variables which bind the result of a store statement, which includes the global variables, are bound to a singleton set in HPT. This singleton set contains a pointer to the abstract location of a store statement. Because each store statement gets only one abstract location in HPT, the variables which bind the result of a store statement are always a singleton set, of which the element refers to a heap location.

Variables which bind the result of an **eval** call hold the union of all heap locations passed to **eval**, but with the F-nodes filtered out. For example, the t variable, which can be a pointer to the abstract location 4, is passed to **eval**. The result of the **eval** call, which is saved in tn, is equal to the nodes storable in location 4 minus the F-nodes.

The formal arguments contain the union of corresponding actual arguments of each function call and thunk. In this example each function is only called once, thus the values of the formal arguments match the values of the actual arguments.

Some variables in the HPT result example do not occur as variables in the program. These variables are *meta variables* and are needed to perform the analysis. There are two classes of meta variables:

- *result variables*, which represent return values of functions with the same name. For example, the *mk* variable holds the resulting nodes of the *mk* function;
- the *apply variable*, which represent all applied arguments to partial applications. This variable is named **apply**. Its use is explained in Section 3.2.1.

The analysis of functions is call insensitive. This means that the results yielded by calls to the same function are jointly represented as a single abstract value. Such a value is bound to a result variable. By convention, a result variable has the same name as its corresponding function. Variables which bind the result of a function call can use the corresponding result variable to retrieve the result of the function call.

3.2.1 First class functions

Haskell allows partially applied functions to be passed as arguments whereas GRIN only allows pointers to nodes to be passed. Partial applied Haskell functions are translated to partial application nodes (P-nodes). Applications to partially applied functions are translated to calls of the function **apply** (see Section 2.3).

As a consequence of the call insensitivity of HPT, HPT estimates the value of a formal argument to be the union of all its corresponding actual arguments. This enforces that all function calls to the same function result in the same estimate. This method works because we statically know which arguments are passed to which functions. However, the existence of partial applications breaks this method because it is no longer statically known which functions are called. For example:

$$
\begin{array}{l}
\mathit{Tuple}\ x\ y = \textbf{unit}\ \ (\#\mathit{CTuple}\ x\ y) \\
\mathit{main} \quad\ = \textbf{store}\ (\#\mathit{CInt}\ 1\ \ \ );\lambda a \rightarrow \\
\qquad\quad\ \ \textbf{store}\ (\#\mathit{CInt}\ 2\ \ \ );\lambda b \rightarrow \\
\qquad\quad\ \ \textbf{unit}\ \ (\#\mathit{P2Tuple});\lambda p \rightarrow \\
\qquad\quad\ \ \textbf{apply}\ p\ a\ \ \ \ \ \ \ ;\lambda g \rightarrow \\
\qquad\quad\ \ \textbf{apply}\ g\ b\ \ \ \ \ \ \ ;\lambda r \rightarrow \\
\qquad\quad\ \ \textbf{unit}\ r \\
\mathit{applymap}: \\
\{\#\mathit{P2Tuple} \rightarrow \#\mathit{P1Tuple} \\
;\#\mathit{P1Tuple} \rightarrow \mathit{Tuple} \\
\}
\end{array}
$$

The above code stores two integer nodes on the heap after which the partially applied function *Tuple* is applied to the two integers. The function *Tuple* returns a tuple containing its arguments. In HPT, the first store statement stores its node in the abstract location 1, the second store statement uses the abstract location 2. Thus the

$$
\begin{array}{ll}
\mathit{AbstractHeap} & = \mathit{Location} \to \mathcal{P}(\mathit{AbstractNode}) \\
\mathit{AbstractEnvironment} & = \mathit{Variable} \to \mathcal{P}(\mathit{AbstractValue}) \\
\mathit{AbstractValue} & = \mathit{AbstractNode} \mid \mathit{AbstractBasicValue} \\
\mathit{AbstractBasicValue} & = \mathcal{P}(\mathit{Location}) \mid \{B\} \\
\mathit{AbstractNode} & = \mathit{Tag}\ [\mathcal{P}(\mathit{AbstractBasicValue})] \\
\mathit{Location} & = \mathbb{N} \\
\mathit{Variable} & = \mathit{String} \\
\mathit{Tag} & = \#\mathit{String}
\end{array}
$$

Figure 3.1: Domain of the abstract values

variables a and b are bound to the abstract values $\{1\}$ and $\{2\}$ while these locations both hold the abstract nodes $\#\mathit{CInt}\ [\{B\}]$. The p variable is bound to the abstract node $\#\mathit{P2Tuple}\ [\,]$, which represent a partial application to the *Tuple* function missing two arguments.

HPT bounds the formal arguments of *Tuple*, x and y, to a union of all corresponding actual arguments. As HPT without support for partial applications cannot find any function call or thunk to *Tuple*, it will falsely conclude that no actuals of *Tuple* are available. Thus without support for partial applications HPT estimates the result of the function *Tuple* to be $\#\mathit{CTuple}\ [\bot, \bot]$, while this should be the value $\#\mathit{CTuple}\ [\{1\}, \{2\}]$.

To support partial applications, HPT uses the applymap. The applymap encodes the flow of control between partial applications nodes and GRIN functions. Hence HPT can estimate which arguments are passed to which functions via **apply** and records this in the **apply** meta variable. This enables the HPT to be call insensitive, even in the presence of partial applications.

In the above example, the first apply statement, **apply** $p\ a$, takes the partial application node $\#\mathit{P2Tuple}\ [\,]$ and applies it to the value $\{1\}$. The applymap encodes that such a call results in a $\#\mathit{P1Tuple}\ [\{1\}]$ node. This value will be bound to the variable g. The second call to apply is the statement **apply** $g\ b$. This statement applies the node $\#\mathit{P1Tuple}\ [\{1\}]$ to the value $\{2\}$. The applymap encodes that such a statement results in a function call to *Tuple*, with the arguments $\{1\}$ and $\{2\}$. This time, not only is the result of the apply call bound to r, but the node $\mathit{FTuple}\ [\{1\}, \{2\}]$ is also added to the apply variable. This F-node has the same name as the function called by the apply statement and the fields of the node contain the actual arguments of that function. Thus, the apply variable records the names of the functions called by the apply statement in an F-node, as well as all the arguments passed to these functions in the fields of the F-node.

Equations of formal arguments use the apply variable to find the actual arguments applied via **apply**. The first formal argument selects the first field of the corresponding F-node from the apply variable, the second formal arguments selects the second field, and so on. Thus, the equation of formal argument x of *Tuple* selects from the apply variable the first field of *FTuple*, the equation of y selects the second field. With the use of the apply variable, HPT does record a value for each formal argument and estimates the result of *Tuple* correctly to be $\#\mathit{CTuple}\ [\{1\}, \{2\}]$.

environment:

	$main\_caf$	$\mapsto$	$\{0\}$
	x_2	$\mapsto$	$\{3\}$
	x_1	$\mapsto$	$\{1\}$
	t	$\mapsto$	$\{4\}$
	tn	$\mapsto$	$\{\#CTuple[\{1\},\{3\}]\}$
	b	$\mapsto$	$\{3\}$
	a	$\mapsto$	$\{1\}$
	x_3	$\mapsto$	$\{3\}$
	f	$\mapsto$	$\{2\}$
	fn	$\mapsto$	$\{\#P1Tuple[\{1\}]\}$
	s_1	$\mapsto$	$\{1\}$
	s_2	$\mapsto$	$\{2\}$
	s_3	$\mapsto$	$\{3\}$
	s_4	$\mapsto$	$\{4\}$
result variables	$Tuple$	$\mapsto$	$\{\#CTuple[\{1\},\{3\}]\}$
	snd	$\mapsto$	$\{\#CInt[\{B\}]\}$
	mk	$\mapsto$	$\{\#CTuple[\{1\},\{3\}]\}$
	$main$	$\mapsto$	$\{\#CInt[\{B\}]\}$
apply variable	$apply$	$\mapsto$	$\{\#FTuple[\{1\},\{3\}]\}$

heap:

0	$\mapsto$	$\{\#CInt[\{B\}],\#Fmain[]\}$
1	$\mapsto$	$\{\#CInt[\{B\}]\}$
2	$\mapsto$	$\{\#P1Tuple[\{1\}]\}$
3	$\mapsto$	$\{\#CInt[\{B\}]\}$
4	$\mapsto$	$\{\#CTuple[\{1\},\{3\}],\#Fmk[\{2\},\{3\}]\}$

Figure 3.2: Example abstract environment and heap

3.3 Equations

HPT computes the approximation by deriving a set of equations from a GRIN program. These equations can either be a combination of join functions ($x \sqcup y$) and select functions ($v \downarrow \#T \downarrow i$), the eval funtion, or the apply function. Equations derived from the running example are shown in Figure 3.3. The eval and apply function are explained in Section 3.3.4. The join and select functions are shown in Figure 3.4.

The select function ($v \downarrow \#T \downarrow i$) is used to access a field from an abstract node value. It selects field i from a node with the tag $\#T$ from the variable v. The select function is the abstract equivalent of a pattern in GRIN. In the example, the variable bindings in the case alternative result in select equations with the scrutinizer as the variable to select from. Another place where the select function is used is the equation of formals. In equations for formal arguments (x_1 in the example) the corresponding actual arguments to partial applications are selected from the apply variable and joined with all statically known arguments.

The join function ($\sqcup$) merges abstract values; nodes are joined based on their tags, so that a tag occurs at most once in an abstract value. For example:

$$\{\#CA[a_1 \ldots a_n], \#CB[b_1 \ldots b_n]\} \sqcup \{\#CA[c_1 \ldots c_n]\}$$

results in $\{\#CA[z_1 \ldots z_n], \#CB[b_1 \ldots b_n]\}$, where $z_i = a_i \sqcup c_i$. The value $\bot$ serves as the identity value: $\bot$ merged with some value a will result in a. To merge locations or tags, the union is taken. The merge function is only defined for arguments which are of the same kind. E.g., it is impossible to merge locations and nodes. The join function is, for example, used to join the results of different case alternatives. It is also used to join the the abstract values of actuals in the equations of formals.

Figure 3.4 shows also the utility function $\mathtt{av}(e)$ which is used to derive equations. This function maps a value expression in GRIN to an abstract value. It is the identity function, except for literals. Literals are all mapped to the abstract value $\{B\}$.

3.3.1 Derivation rules

This section describes the derivation of the equations from a GRIN program. Boquists Ph.D. thesis [6] ignores this process and shows only the results of HPT; one of his earlier papers [7] describes the derivation process only informally. This lack of description is our motivation for a more formal approach.

The derivation of the equations are described by a set of derivation rules. These rules are of the following form:

$$\textsc{Rule name}: \frac{\begin{array}{l} prerequisite_1 \\ \ldots \\ prerequisite_n \end{array}}{consequence}$$

This means that if all $prerequisite_i$ are proven, the *consequence* can be concluded. To save some space, small prerequisites are placed side by side, separated by semicolons

identifier equations ($\mathcal{E}$):

$$
\begin{aligned}
main\_caf &= \{0\} \\
x_1 &= \bigsqcup\{\} \sqcup apply \downarrow \#FTuple \downarrow 1 \\
x_2 &= \bigsqcup\{\} \sqcup apply \downarrow \#FTuple \downarrow 2 \\
t &= \bigsqcup\{s_4\} \sqcup apply \downarrow \#Fsnd \downarrow 1 \\
tn &= \texttt{eval}\ t \\
a &= tn \downarrow \#CTuple \downarrow 1 \\
b &= tn \downarrow \#CTuple \downarrow 2 \\
f &= \bigsqcup\{s_2\} \sqcup apply \downarrow \#Fmk \downarrow 1 \\
x_3 &= \bigsqcup\{s_3\} \sqcup apply \downarrow \#Fmk \downarrow 2 \\
fn &= \texttt{eval}\ f \\
s_1 &= \{1\} \\
s_2 &= \{2\} \\
s_3 &= \{3\} \\
s_4 &= \{4\} \\
alt1 &= \texttt{eval}\ b \\
Tuple &= \#CTuple[x_1, x_2] \\
snd &= \bigsqcup\{alt1\} \\
mk &= \texttt{apply}\ fn\ x_3 \\
main &= snd \\
apply &= \bot
\end{aligned}
$$

heap equations ($\mathcal{H}$):

$$
\begin{aligned}
0 &= \#Fmain[] \sqcup main \\
1 &= \#CInt[\{B\}] \sqcup \bot \\
2 &= \#P1tuple[s1] \sqcup \bot \\
3 &= \#CInt[\{B\}] \sqcup \bot \\
4 &= \#Fmk[s2, s3] \sqcup mk
\end{aligned}
$$

Figure 3.3: Example equations

$$
\begin{array}{rcll}
\{\ldots \#T[x_1 \ldots x_n] \ldots\} \downarrow \#T \downarrow i & = & x_i & \text{when } 1 \leq i \leq n \\
\{\ldots \#T[x_1 \ldots x_n] \ldots\} \downarrow \#T \downarrow i & = & \bot & \text{otherwise} \\
\{\ldots \#T[x_1 \ldots x_n] \ldots\} \downarrow \#S \downarrow i & = & \bot & \\
 & & & \\
\{\ldots \#T[x_1 \ldots x_n] \ldots\} \sqcup \{\ldots \#S[y_1 \ldots y_n] \ldots\} & = & \{\ldots \#T[x_1 \ldots x_n], \#S[y_1 \ldots y_n] \ldots\} & \\
\{\ldots \#T[x_1 \ldots x_n] \ldots\} \sqcup \{\ldots \#T[y_1 \ldots y_n] \ldots\} & = & \{\ldots \#T[x_1 \cup y_1 \ldots x_n \cup y_n] \ldots\} & \\
x \sqcup \bot & = & x & \\
x \sqcup y & = & x \cup y & \\
\mathtt{av}((\mathrm{T}\ x_1\ \ldots\ x_n)) & = & \mathtt{av}(T)[\mathtt{av}(x_1) \ldots \mathtt{av}(x_n)] & \\
\mathtt{av}(var) & = & var & \\
\mathtt{av}(Tag) & = & Tag & \\
\mathtt{av}(literal) & = & \{B\} &
\end{array}
$$

$\#T$ and $\#S$ represent different tags.

Figure 3.4: Utility functions

(;). The prerequisites and consequence are structured in the form of judgements. These judgements have the following form:

$$context \overset{\text{judgement type}}{\vdash} [\![code]\!] : results$$

The values passed as context and the values returned as results can be the following:

symbol	**domain**	**represents**
$\mathcal{E}$	$\{Variable \rightarrow Equation\}$	identifier equations
$\mathcal{H}$	$\{Location \rightarrow Equation\}$	heap equations
$\mathcal{A}$	$\{f \mapsto a_1 \ldots a_n\}$	actual function arguments
$\mathcal{R}$	$\{Tag \mapsto v\}$	result variables
v	$Variable$	a variable
$(\mathrm{T}\ a_1\ \ldots\ a_n)$	$(Tag\ \mathcal{P}(Variable))$	a node
p	$Variable \mid (Tag\ \mathcal{P}(Variable))$	a variable or node

A judgement has a type, which describes which contextual information is expected and which results are returned by that judgement. The following table list the possible judgement types:

type	**context**	**results**
module		$\mathcal{E}$, $\mathcal{H}$
resVars		$\mathcal{R}$
bind	$\mathcal{R}$, $\mathcal{A}$	$\mathcal{E}$, $\mathcal{H}$, $\mathcal{A}$
globals	$\mathcal{R}$	$\mathcal{E}$, $\mathcal{H}$
expr	$\mathcal{R}$, p	$\mathcal{E}$, $\mathcal{H}$, $\mathcal{A}$

$$\textsc{Module}: \frac{\begin{array}{l} \overset{\text{resVars}}{\vdash} [\![evalMapping]\!] : \mathcal{R} \\ \mathcal{R} \overset{\text{globals}}{\vdash} [\![globalVars]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A} \\ \mathcal{R}, \mathcal{A} \overset{\text{bind}}{\vdash} [\![binds]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A} \end{array}}{\overset{\text{module}}{\vdash} \left[\!\!\left[\begin{array}{l} \texttt{module}\ n \\ \quad \{globalVars\} \\ \quad \{binds\} \\ \quad \texttt{evalmap}\ \{evalMapping\} \\ \quad \texttt{applymap}\ \{applyMapping\} \end{array} \right]\!\!\right] : \mathcal{E}, \mathcal{H}}$$

Figure 3.5: Module

The derivation rules are declarative. The membership and subset relations found in the prerequisites of derivation rules can be translated into computational actions. In the $\textsc{ResVar}_{unit}$ rule defined in Figure 3.6, for example, a value must be member of the rule result value $\mathcal{R}$ which can be turned into a computation by adding the value to $\mathcal{R}$. When $\mathcal{R}$ is passed as context of a consequence, as in Figure 3.8 in the Store rule, a membership relation of the prerequisites translates to a selection from $\mathcal{R}$. When judgements return the same result variables, these results are joined implicitly. An example is the Module rule in Figure 3.5 where, among others, $\mathcal{A}$ is returned by both the bind and the global judgements and passed as context to the bind judgement. This is translated to a join of the resulting $\mathcal{A}$'s of both judgements which is then passed to the bind judgment.

The top level derivation rule is given in Figure 3.5. This derivation rule does not use any contextual information and derives a set of identifier equations $\mathcal{E}$ and a set of store equations $\mathcal{H}$ from a whole program by dividing the program into pieces and passing those pieces to rules which derive equations of parts of the program.

Besides the resulting equations, the derivation process uses two maps: the result variables map ($\mathcal{R}$), and the actual arguments map ($\mathcal{A}$).

The result variables map encodes whether a node represents a thunk or not. The result variables map is generated from the evalmap, shown in Figure 3.6, and does not depend on any contextual information. Each tag in the evalmap is translated to an element in the result variable map: if the tag maps to a **unit** element in the evalmap, the tag represent a value in WHNF and maps to $\perp$ in the result variables map; if the tag maps to a function identifier, the tag represents a thunk and the tag in the result variables map maps to the same function identifier. The result variable map is thus the same as the eval map, but with the **unit** written as $\perp$ and without the node size information.

The actual arguments map is a mapping from function identifiers to all actual parameters of calls to that function; both direct calls and thunks are stored in this map. The actual arguments map is used by the derivation of equations of the formal arguments which needs to join all their corresponding actual arguments. The actual arguments map is created in Figure 3.8 in the Store rule and in Figure 3.9 in the Call rules.

$$\textsc{ResVars} : \frac{\overset{\text{resVars}}{\vdash} [\![evalMap_1]\!] : \mathcal{R}; \ldots; \overset{\text{resVars}}{\vdash} [\![evalMap_n]\!] : \mathcal{R}}{\overset{\text{resVars}}{\vdash} [\![evalMap_1; \ldots; evalMap_n]\!] : \mathcal{R}}$$

$$\textsc{ResVar}_{unit} : \frac{(T \mapsto \bot) \in \mathcal{R}}{\overset{\text{resVars}}{\vdash} [\![(T, size) \rightarrow \texttt{unit}]\!] : \mathcal{R}}$$

$$\textsc{ResVar}_{var} : \frac{(T \mapsto v) \in \mathcal{R}}{\overset{\text{resVars}}{\vdash} [\![(T, size) \rightarrow v]\!] : \mathcal{R}}$$

Figure 3.6: Result variables

The actual arguments map of the running example is:

$$\mathcal{A} = \{(snd \mapsto s4), (mk \mapsto s2\ s3)\}$$

This actual arguments map records no arguments for the *Tuple* function, as this function is not called, and no closures are created.

3.3.2 Formal arguments

An equation for the formal arguments of a function joins all its corresponding actual arguments with the actuals selected by the apply variable (see Section 3.2.1). The derivation of such an equation is shown in Figure 3.7. The actual arguments map ($\mathcal{A}$) is passed as context to this rule. All actuals of a function are selected from this actual arguments map. This results in a set of actuals, refered to by $args_i$, for the formal argument a_i. Actual arguments passed by the use of **apply** are selected from the apply variable. The selected values from the apply variable are then joined with the join of all actual arguments.

Note that the actual arguments map occurs both as context and as result of the derivation rule. This is best explained by looking at the BIND rule as a computation. The computation described by the rule result in an actual arguments map which hold only the actuals of a single binding. This map is passed upwards so that the all actual arguments can be joined at the top level. This complete actual arguments map is then passed down again (as context) and can be used to find all actual arguments.

3.3.3 Store statements

Store statements come in two flavors. The store statements of the global variables and the store statements in the bodies of functions. Both are treated the same in HPT.

$$\textsc{Bind}: \frac{\begin{array}{l}\mathcal{R}, foo \overset{\text{expr}}{\vdash} [\![e]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A} \\ \left\{ \begin{array}{l} a_1 = \bigsqcup args_1 \sqcup \textbf{apply} \downarrow \text{Ffoo} \downarrow 1 \\ \ldots \\ a_n = \bigsqcup args_n \sqcup \textbf{apply} \downarrow \text{Ffoo} \downarrow n \end{array} \right\} \subseteq \mathcal{E} \\ (foo \mapsto args_1 \ldots args_n) \subseteq \mathcal{A} \end{array}}{\mathcal{R}, \mathcal{A} \overset{\text{bind}}{\vdash} [\![foo\ a_1\ \ldots\ a_n = e]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

$$\textsc{Binds}: \frac{\mathcal{R}, \mathcal{A} \overset{\text{bind}}{\vdash} [\![b_1]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}; \ldots; \mathcal{R}, \mathcal{A} \overset{\text{bind}}{\vdash} [\![b_n]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}{\mathcal{R}, \mathcal{A} \overset{\text{bind}}{\vdash} [\![b_1; \ldots; b_n]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

Figure 3.7: Bindings

Figure 3.8 shows the derivation rules. The GlobalVar rule calls the Store rule for each global variable and returns the derived $\mathcal{E}$ and $\mathcal{H}$ equations as well as the found actual arguments. The Store rule is the first rule of the judgement type *expr*. All the rules on GRIN expression are of the expr judgement type. These rules need a result variable map and a *target pattern*. The target pattern denotes a variable or a pattern with multiple variables to which the identifier equations must be bound. These rules return the triple $\mathcal{E}$, $\mathcal{H}$, and $\mathcal{A}$.

The result of a store statement is an identifier equation and a heap equation. Each store statement is given an unique location (*loc*) in the abstract heap. The identifier equation binds the target variable to this unique location. The abstract heap equation consist of an abstract node joined by a result variable. The abstract node is the abstract variant of the node stored by the store statement. In an actual run of the program the node on the heap might be updated. This only happens when the node represents a thunk: the node will be overwritten with the result of the function call when the thunk is evaluated. In the analysis this behaviour is accomplished by joining the result variable of that function call with the node stored by the store statement.

3.3.4 Function calls

Derivation of equations of a function call is shown in Figure 3.9. While the function call only modifies $\mathcal{E}$ and $\mathcal{A}$, the $\mathcal{H}$ is also returned. The rules leave out $\emptyset \subseteq \mathcal{H}$.

The result of a function is stored in its result variable (see Section 3.2). The result of a function call is simply a copy of that value. The $\textsc{Call}_{node}$ rule gets a target pattern. For each variable in this target pattern it selects a field from the result variable.

Two functions, **apply** and **eval** are treated differently. The **apply** function is treated different as it adds information to the **apply** variable beside computing the result of the equation. The **eval** function is treated different because all other functions call **eval** to evaluate their arguments and thus all heap locations are accessed by **eval**. This

$$\textsc{GlobalVar}: \frac{\begin{array}{l}\mathcal{R}, v_1 \overset{\text{expr}}{\vdash} [\![\texttt{store}_{loc_1}\ (\mathrm{T}_1\ a_{11_1}\ \ldots\ a_{1n_1})]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A} \\ \ldots \\ \mathcal{R}, v_m \overset{\text{expr}}{\vdash} [\![\texttt{store}_{loc_m}\ (\mathrm{T}_\mathrm{m}\ a_{m1_m}\ \ldots\ a_{mn_m})]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}\end{array}}{\mathcal{R} \overset{\text{globals}}{\vdash} \left[\!\!\left[\begin{array}{l} v_1 \leftarrow \texttt{store}_{loc_1}\ (\mathrm{T}_1\ a_{11_1}\ \ldots\ a_{1n_1}) \\ \ldots \\ v_m \leftarrow \texttt{store}_{loc_m}\ (\mathrm{T}_\mathrm{m}\ a_{m1_m}\ \ldots\ a_{mn_m}) \end{array}\right]\!\!\right] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

$$\textsc{Store}: \frac{\begin{array}{l} av_1 = \texttt{av}(a_1) \\ \ldots \\ av_n = \texttt{av}(a_n) \\ (T \mapsto rv) \in \mathcal{R} \\ (rv \mapsto av_1 \ldots av_n) \in \mathcal{A} \\ v = \{loc\} \in \mathcal{E} \\ loc = (\mathrm{T}\ av_1\ \ldots\ av_n) \sqcup rv \in \mathcal{H} \end{array}}{\mathcal{R}, v \overset{\text{expr}}{\vdash} [\![\texttt{store}_{loc}\ (\mathrm{T}\ a_1\ \ldots\ a_n)]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

Figure 3.8: Store statements

means that if **eval** would be treated as call insensitive, like other functions, HPT would result in the estimation that all heap locations of the program are a possible argument to **eval**, which is as bad as not analysing **eval** at all.

HPT uses two special equations to analyse **eval** and **apply**. These equations have the appropriate names *apply* and *eval*.

Eval

The real eval function has one argument: a pointer. Eval fetches the contents of this pointer from the heap and, when the fetched node represents a thunk, calls the corresponding function and update the node with the result.

However, the abstract heap equations of the store statements already take into account the update after the evaluation of a thunk. The abstract eval function can therefore simply fetch the contents of its argument from the abstract heap. This abstract location contain the thunk and the result of the function call. However, thunks will never be returned by the eval function, thus the abstract eval equation filters them from the result.

Apply

For each apply call the abstract apply equation returns a node representing the result. Such a node can be one of the following:

$$\textsc{Eval}: \frac{v = \texttt{eval}\ w \in \mathcal{E}}{\mathcal{R}, v \overset{\text{expr}}{\vdash} [\![\texttt{eval}\ w]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

$$\textsc{Apply}: \frac{v = \texttt{apply}\ w\ u_1\ \ldots u_n \in \mathcal{E}}{\mathcal{R}, v \overset{\text{expr}}{\vdash} [\![\texttt{apply}\ w\ u_1\ \ldots u_n]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

$$\textsc{Call}_{node}: \frac{\left\{ \begin{array}{lcl} a_1 & = & foo \downarrow \mathrm{T} \downarrow 1 \\ \ldots & & \\ a_n & = & foo \downarrow \mathrm{T} \downarrow n \end{array} \right\} \subseteq \mathcal{E}; (foo \mapsto x_1 \ldots x_n) \in \mathcal{A}}{\mathcal{R}, (\mathrm{T}\ a_1\ \ldots\ a_n) \overset{\text{expr}}{\vdash} [\![foo\ x_1 \ldots x_n]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

$$\textsc{Call}_{var}: \frac{v = foo \in \mathcal{E}; (foo \mapsto x_1 \ldots x_n) \in \mathcal{A}}{\mathcal{R}, v \overset{\text{expr}}{\vdash} [\![foo\ x_1 \ldots x_n]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

Figure 3.9: Function calls

- A P-node with the fields containing the arguments already applied with the newly applied argument appended. This P-node represent a partial application which takes one argument less than the P-node passed to apply.
- The result of a function call when the newly applied arguments happens to be the last argument of the application.

Each time the apply equation is calculated a node is added to the **apply** variable. As explained in Section 3.2.1 this node stores the correlation between the function represented by the P-node passed to apply and the actual arguments of that function.

The actual **apply** equation can take a list of arguments which should be applied. The arguments are applies one by one as described above.

3.3.5 Foreign function interface

The foreign function interface (ffi) statement is used for two different purposes: the implementation of primitives, and communication to the outside world. Primitives are known to the compiler. The equations for primitives are the nodes which might be returned by them. E.g., the *primEqInt* primitive returns (`#`*CTrue*) or (`#`*CFalse*).

Communication to the outside world is a bit more problematic: there is no information availible on what these external functions return. The type information of Haskell can give us a set of possible data constructors, and thus tags. But in the current implementation the ffi statements are limited to primitives or functions which return a basic integer.

$$\text{FFI}_{prim} : \frac{v = \texttt{av}(primFoo) \in \mathcal{E}}{\mathcal{R}, v \overset{\text{expr}}{\vdash} [\![\textbf{ffi}\ primFoo\ x_1 \ldots x_n]\!], \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

$$\text{FFI}_{external} : \frac{v = \{B\} \in \mathcal{E}}{\mathcal{R}, v \overset{\text{expr}}{\vdash} [\![\textbf{ffi}\ foo\ x_1 \ldots x_n]\!], \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

Figure 3.10: Foreign function interface

The derived equations for ffi statements are shown in Figure 3.10. The function $\texttt{av}(\ldots)$ is used to represent the lookup of which nodes the given primitive might return.

3.3.6 Control flow

Deriving equations for control flow is shown in Figure 3.11. There are two types of control flow expressions: the sequence operator, and the case statement. The sequence operator does not generate equations by itself. It merely passes the pattern defined by it to the statement which should bind its result to the variables from that pattern.

The case statement binds each variable of a pattern to the value selected from the scrutinized variable. For example, the first field of the first alternative selects the first field of the node with tag T_1 from the scrutinized variable w. The result of each alternative is joined to the variable after a case. To be able to express the join of each alternative as an equation in HPT, all alternatives must end with an **unit** statement which returns a single variable. This is enforced by a transformation before deriving the equations.

3.3.7 Unit statements

The derivation of an unit statement comes in four variants: the target pattern can be a node or a variable and the argument of the unit statement can be a node or a variable. These variants are shown in Figure 3.12. A node unit statement which matches a node target pattern each field variable of the pattern is bound to the value of the node in the unit statement. The tag of the nodes in the target pattern and the unit statement is assumed to be the same. When a node unit statement matches a variable target pattern, this variable is bound to the complete node.

A variable unit statement which matches a node target pattern is comparable to a case statement with a single alternative. Each field of the pattern selects the corresponding field from the nodes bound to the variable in the unit statement. When a variable unit statement matches a variable target pattern, the variables become aliases for each other thereby holding the same value.

$$\textsc{Case}: \frac{\left\{\begin{array}{lcl} a_{11} & = & w \downarrow \mathrm{T}_1 \downarrow 1 \\ \ldots & & \\ a_{1n} & = & w \downarrow \mathrm{T}_1 \downarrow n \\ \ldots & & \\ a_{m1} & = & w \downarrow \mathrm{T}_\mathrm{m} \downarrow 1 \\ \ldots & & \\ a_{mn} & = & w \downarrow \mathrm{T}_\mathrm{m} \downarrow n \\ v & = & \bigsqcup\{x_1 \ldots x_m\} \end{array}\right\} \subseteq \mathcal{E};\ \begin{array}{l} \mathcal{R}, x_1 \overset{\text{expr}}{\vdash} [\![e_1]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A} \\ \ldots \\ \mathcal{R}, x_m \overset{\text{expr}}{\vdash} [\![e_m]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A} \end{array}}{\mathcal{R}, v \overset{\text{expr}}{\vdash} \left[\!\!\left[\begin{array}{l} \texttt{case}\ w\ \texttt{of} \\ \quad (\mathrm{T}_1\ a_{11_1}\ \ldots\ a_{1n_1}) \rightarrow \\ \quad\quad e_1; \lambda x_1 \rightarrow \texttt{unit}\ x_1 \\ \quad \ldots \\ \quad (\mathrm{T}_\mathrm{m}\ a_{m1_m}\ \ldots\ a_{mn_m}) \rightarrow \\ \quad\quad e_m; \lambda x_m \rightarrow \texttt{unit}\ x_m \end{array}\right]\!\!\right] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

$$\textsc{Seq}: \frac{\mathcal{R}, p \overset{\text{expr}}{\vdash} [\![e_1]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}; \mathcal{R}, q \overset{\text{expr}}{\vdash} [\![e_2]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}{\mathcal{R}, q \overset{\text{expr}}{\vdash} [\![e_1; \lambda p \rightarrow e_2]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

Figure 3.11: Flow control

$$\textsc{Unit}_{node \rightarrow node}: \frac{\left\{\begin{array}{lcl} a'_1 & = & \texttt{av}(a_1) \\ \ldots & & \\ a'_n & = & \texttt{av}(a_n) \end{array}\right\} \subseteq \mathcal{E}}{\mathcal{R}, (\mathrm{T}\ a'_1\ \ldots\ a'_n) \overset{\text{expr}}{\vdash} [\![\texttt{unit}\ (\mathrm{T}\ a_1\ \ldots\ a_n)]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

$$\textsc{Unit}_{node \rightarrow var}: \frac{v = \texttt{av}((\mathrm{T}\ a_1\ \ldots\ a_n)) \in \mathcal{E}}{\mathcal{R}, v \overset{\text{expr}}{\vdash} [\![\texttt{unit}\ (\mathrm{T}\ a_1\ \ldots\ a_n)]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

$$\textsc{Unit}_{var \rightarrow node}: \frac{\left\{\begin{array}{lcl} a_1 & = & \texttt{av}(v) \downarrow \#T \downarrow 1 \\ \ldots & & \\ a_n & = & \texttt{av}(v) \downarrow \#T \downarrow n \end{array}\right\} \subseteq \mathcal{E}}{\mathcal{R}, (\mathrm{T}\ a_1\ \ldots\ a_n) \overset{\text{expr}}{\vdash} [\![\texttt{unit}\ v]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

$$\textsc{Unit}_{var \rightarrow var}: \frac{v = \texttt{av}(w) \in \mathcal{E}}{\mathcal{R}, v \overset{\text{expr}}{\vdash} [\![\texttt{unit}\ w]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}}$$

Figure 3.12: Unit statement

3.3.8 Normalizing

Not all combinations of patterns and GRIN constructs are covered. For example, the case statement expect expects every alternative to end with a unit statement which returns a single variable. And the function rules for **eval** and **apply** are only defined for a singe variable as target pattern. A transformation is applied on the program which add extra unit statements so that the derivation rules defined in this section cover all possible constructs of valid GRIN input.

3.4 Solving the equations

The final step of HPT is to solve the derived equations. This is done by a fixpoint computation. A fixpoint computation is an iterative process. Each iteration the values of the equations are calculated based on the values of the previous iteration. Once all equations hold the same values as the previous iteration, all further iterations are guaranteed to result in the same values. At this point all equations hold.

To start the first iteration, all variables and abstract heap locations are bound to the value bottom ($\bot$). Each time an equation refers to some variable the value of the previous iteration is used. For example, the equation $Tuple \mapsto \#CTuple[x1, x2]$ results in the value $\#CTuple[\bot, \bot]$. The new values are compared to the values of the previous iteration, if any value is changed a new iteration is started and the values of the equations are calculated on the current values.

The current implementation does not always use the values of the previous iteration, instead it uses the value currently recorded. If the equation is not yet calculated in the current iteration, the value of the previous iteration is used, once it is calculated, that value is used to calculated the other equations in the current equation.

Chapter 4

Exceptions

Exceptions fall into two categories: *synchrone* and *asynchrone* exceptions. Synchrone exceptions consist of failures of computations or actions of the program. Examples are division by zero, indices which are out of the bounds of an array, or accessing unreadable files. Asynchrone exceptions, or interrupts, are generated by external events. Examples of external events are the Unix signals. These are signals which the operating system can send to a program, for example when the user presses the control+C key combination. When a program receives an Unix signal, it stops its current computation and handles the recieved signal. When the signal is handled, the program might resume (or restart) its computation.

This chapter adds support for handling of exceptions in GRIN. Although the construct might prove useful for asynchrone exceptions only synchrone exceptions are implemented here.

4.1 Exception handling in Haskell

A valid question is whether GRIN needs special exception handling support. To answer that question we look at the exception support as modelled in Haskell. In Haskell, a failure of a computations can be implemented in different ways. For example, one might implement exceptions explicitly in a datatype:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Value}\ a = \mathit{OverFlowException} \mid \mathit{OK}\ a \\
\mathit{add3}\ x\ y\ z = \textbf{case}\ x + y\ \textbf{of} \\
\qquad\qquad \begin{array}{ll} \mathit{OverFlowException} & \rightarrow \mathit{OverFlowException} \\ \mathit{OK}\ a & \rightarrow a + z \end{array}
\end{array}
$$

The *add3* function test the result of the addition of x and y. When the addition has failed, the exception is propagated to the caller of *add3*, if it is successfull z is added to the result of the first addition. This is known as the test-and-propagate method.

A big drawback of the test-and-propagate method is that every function result must be tested explicitly on failures, which makes a program much harder to read: even a simple addition of three values, as in the example above, becomes obfuscated with a case expression. In some parts of Haskell, the test-and-propagate code can be hidden inside

a library. This is the case for I/O computations, which are implemented in Haskell by using monads to force sequential evaluation of I/O operations. For exceptions outside monads, like the *add3* example above, the test-and-propagate method creates a bigger problem than readability: the case statement fixes the evaluation order, which takes away optimisation possibilities from the compiler.

The Haskell 98 standard [12] circumvents the problem of exceptions outside monads by defining only exceptions in I/O computations. But an extension of Haskell exists, called imprecise exceptions [15, 16], which introduces a semantics for exceptions outside monads without fixing the evaluation order. With the semantics of imprecise exceptions, the program does not explicitly propagate exceptions. For example, the *add3* function is defined in the program as:

$$add3\ x\ y\ z = x + y + z$$

The compiler must ensure that the overflow exception which might occur in the addition of two values is passed to the exception handler. With the semantics of imprecise exceptions the compiler is free to transform the *add3* function defined in the program to the former definition of *add3* or to, for example, the definition:

$$\begin{aligned} add3\ x\ y\ z = \textbf{case}\ & y + z\ \textbf{of} \\ & OverFlowException \rightarrow OverFlowException \\ & OK\ a \rightarrow x + a \end{aligned}$$

The programmer has no guarantees which exception has occured. The only guarantee is that one of the exceptions which might be triggered by the expression is passed to the exception handler.

The flow of the exception from location of occurence to the exception handler can be implemented in the compiler by test-and-propagate. However, the test-and-propagate method suffers from performance loss: each callsite between the exception and the handler needs code to test-and-propagate, which makes the program both bigger and slower. When a language contains constructs to jump directly back to the exception handler no propagation is needed, and thus the perfomance impact is non-existent. It is this jumping to the function in which the exception handler is defined which needs special support in GRIN.

4.2 Exception handling in GRIN

Modern imperative languages define an exception handler with a statement in the form of:

try{
  guardedCode
}**catch** (*e*){
  exceptionHandler
}

or something very similar. Code in the brackets after the **try** keyword are guarded for exceptions. Excpetions thrown in this guarded code are passed to the exception

exp	+=	**try** *exp* **catch**(*var*) *exp*	catch exception
sexp	+=	**throw** *val*	throw exception

Figure 4.1: GRIN exception syntax

handler, which is defined by the code in the brackets after the **catch** keyword. The exception passed to the exception handler is bound to the variable defined in the **catch** keyword, in this example e.

If an exception occurs inside the guarded code, the control is passed to the active exception handler. When try-catch statements are nested, the innermost exception handler is active. Whenever the control leaves the guarded code of a try-catch statement, the previous exception handler becomes active again.

This model of exception handling is enough to encode exceptions in Haskell, including the imprecise exceptions extension. We adopt this exception handling model in GRIN. The syntax for the new statements is shown in Figure 4.1.

4.3 Compiling to GRIN

First, let we look at a small example of exceptions in Haskell with the imprecise exceptions extension:

$$
\begin{aligned}
&\mathit{thr}\ x = \textbf{throw}\ x \\
&\mathit{main} = \textbf{catch}\ (\mathit{thr}\ \mathit{DivByZero} + \mathit{thr}\ \mathit{OutOfBounds}) \\
&\qquad\qquad (\lambda e \rightarrow 1)
\end{aligned}
$$

This program performs the addition of two expression which both raise an exception. When an exception is thrown the exception handler is executed. But which of the two exceptions is thrown is decided by the compiler.

The front-end compiler can express this example in GRIN as follows:

$$
\begin{aligned}
&\mathit{add}\ a\ b = \{\textbf{eval}\ a; \lambda(\#\mathit{CInt}\ a') \rightarrow \\
&\qquad\qquad \textbf{eval}\ b; \lambda(\#\mathit{CInt}\ b') \rightarrow \\
&\qquad\qquad \textbf{ffi}\ \mathit{primAddInt}\ a'\ b'; \lambda(\#U\ c') \rightarrow \\
&\qquad\qquad \textbf{unit}\ (\#\mathit{CInt}\ c') \\
&\qquad\qquad \} \\
&\mathit{thr}\ x \quad = \{\textbf{throw}\ x \\
&\qquad\qquad \} \\
&\mathit{main} \quad = \{\textbf{try}\{ \\
&\qquad\qquad\quad \textbf{store}\ (\#\mathit{CDivByZero}); \lambda e_1 \rightarrow \\
&\qquad\qquad\quad \textbf{store}\ (\#\mathit{COutOfBounds}); \lambda e2 \rightarrow \\
&\qquad\qquad\quad \textbf{store}\ (\#\mathit{Fthr}\ e_1); \lambda f1 \rightarrow \\
&\qquad\qquad\quad \textbf{store}\ (\#\mathit{Fthr}\ e2); \lambda f2 \rightarrow \\
&\qquad\qquad\quad \mathit{add}\ f1\ f2; \\
&\qquad\qquad \}\textbf{catch}\ (e)\{
\end{aligned}
$$

```
    unit (#CInt 1)
  }; λr →
  unit r
}
```

This example shows some design choices for mapping Haskell exceptions to GRIN. To summarize:

- A pointer to the exception is thrown, rather than the node itself;
- The catch statement is not the last statement of a function body.

The first item is discussed in Section 4.3.1. The last item simplifies the code generator. The code generator must emit different code for the last statement of a function body: the result of the last statement is used as the return value of a function. However, if the last statement is a **catch** statement, the previous exception handler must be restored before returning from a function. This means that the code generator must emit code to temporary hold the result while restoring the previous exception handler. When the **catch** statement happens to be the last statement we append a unit statement after it.

4.3.1 Sharing of exceptions

Functions in Haskell are free of side effects. This fact is used to share the result of common computations. To accomplish this, a thunk is overwritten by its result when it is evaluated. Exceptions can also be shared: when the evaluation of a thunk throws an exception, it always throws the same exception.[1] In many cases it will be more efficient to throw the exception as soon as the thunk is evaluated, instead of evaluating the thunk again.

To implement sharing of exceptions, the **eval** function must be modified. An *Fthrow*-node is used to represent a suspended throw statement. When this node is passed to **eval** the exception represented by the *Fthrow*-node is thrown. Furthermore, each F-node under evaluation must be updated in the case of an exception. This is done by installing an exception handler which updates the F-node and re-throws the exception. An example of the new **eval** function is given in Figure 4.2.

The **update** statement in the exception handlers of **eval** (as shown in Figure 4.2) updates the thunk with a *Fthrow*-node. The field of this node is the variable defined in the **catch** statement. Since a node cannot contain another node, only a pointer to another node, the variable defined in the **catch** statement must be a pointer. As a result of this, **throw** statements must throw a pointer to an exception node rather than the exception node itself.

4.3.2 Implementation details

Any exception not caught by the GRIN program is caught by the startup code, which displays a non informative message. In the future, this code might be added to the GRIN program as a new main, which enables the default exception handler to do more than just print "an error has occured."

[1] This is *not* the case with asynchrone exceptions. They are outside influences.

```
eval p = {fetch p; λn →
          case n of
            { (Fadd a b) → {try{
                              add a b; λr →
                              update p r; λ() →
                              unit r
                            }catch (e){
                              update p (Fthrow e); λ() →
                              throw e
                            }; λx →
                            unit x
                           }
            ; (Fthrow e) → {throw e
                           }
            ...
            }
          }
```

Figure 4.2: Eval with exception sharing

The current version of EH and the implementation of EHC have no notion of exceptions. To use the exception constructs in EH, we define the throw and catch functions as foreign functions. These foreign functions are transformed by the GRIN compiler into the try-catch and throw statements.

4.4 Semantics

To express the semantics of exceptions the semantic function for expressions ($\mathcal{E}$) is modified as shown in Figure 4.3. The function takes an extra argument, the *mode argument*, and returns an extra value, the *mode value*. The mode argument and value are usually the empty set ($\emptyset$). This is called the *normal mode* of the semantic function. When an exception is thrown, the function enters *exception mode* and the mode argument and value hold a pointer to the exception which is thrown.

In normal mode the semantic function is as previously defined in Section 2.4 with the addition of the mode argument and value. The mode argument is returned as the mode result without changing its value. In exception mode the semantic function is the identity function (first rule of Figure 4.3). The **throw** statement switches from normal mode to exception mode. Its mode return value is the pointer to the exception. When the body of the **catch** statement returns in exception mode, the pointer to the exception is bound to the argument of the exception handler after which the semantic function (in normal mode) is applied to the exception handler.

4.5 HPT Analysis

The HPT is modified. It must analyze the flow of exceptions, and the variable bound by the catch statement, called the *catch variable*, needs an environment equation. This

Semantic function:

$$\mathcal{E} :: exp \rightarrow Env \rightarrow Store \rightarrow Mode \rightarrow (Value, Store, Mode)$$

Semantic domain:

$$Mode = Loc \mid \emptyset$$

Additional expression semantics:

$$\begin{aligned}
&\mathcal{E}\,[m\quad\quad]\,\rho\,\sigma\,l = (\rho, \sigma, l)\\
&\mathcal{E}\,[\textbf{throw}\ e]\,\rho\,\sigma\,\emptyset = \textbf{let}\ l = \mathcal{V}\,[e]\,\rho\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\rho, \sigma, l)\\
&\mathcal{E}\,[\textbf{try}\quad\quad\{m\}\\
&\quad\ \textbf{catch}\ (e)\{k\ \}\\
&\quad\ ]\,\rho\,\sigma\,\emptyset \qquad\quad = \textbf{let}\ (\rho', \sigma', z) = \mathcal{E}\,[m]\,\rho\,\sigma\,\emptyset\\
&\qquad\qquad\qquad\qquad\quad\ \ \rho'' \qquad\quad = \rho'\,[e \mapsto z]\\
&\qquad\qquad\qquad\qquad\quad \textbf{in if}\ z \equiv \bot\\
&\qquad\qquad\qquad\qquad\qquad\ \ \textbf{then}\ (\rho', \sigma', z)\\
&\qquad\qquad\qquad\qquad\qquad\ \ \textbf{else}\ \mathcal{E}\,[k]\,\rho''\,\sigma'\,\emptyset
\end{aligned}$$

Figure 4.3: GRIN Exceptions Semantics

catch variable holds all pointers to exceptions which might be thrown by the statements inside the scope of catch. The equation of the catch variable is a join of all pointers which are thrown as exceptions in the scope of the exception handler. To collect all variables which hold pointers to exceptions the expression derivation rules are extended with an exceptions set ($\mathcal{X}$). The domain of exceptions set ranges over variables. The type of the expression derivation rules (expr) are changed. The expr type includes the exceptions set as its result:

type	**context**	**results**
expr	$\mathcal{R}$, p	$\mathcal{E}$, $\mathcal{H}$, $\mathcal{A}$, $\mathcal{X}$

Figure 4.4 shows the changes in the derivation rules of HPT. Note that the application of the rules can result in more than one instance of an exception set, each with its own scope. This is different than the other sets used by the derivation rules, for which only one instance exists. A new exception set is created by two rules. The BIND rule introduces a new exception set to its body. The CATCH rule creates an exception set to record all variables which hold the exceptions thrown in the guarded statements.

The **throw** statement adds its argument to the exceptions set. The catch statement adds an environment equation: it binds the catch variable to the join of all the variables in the exception set of the body of the **catch** statement.

Exceptions which pass the bounds of functions are collected in the same way as function results. For each function an *exception result variable* is introduced. All exceptions thrown but not catched in a single function are stored in the exception result variable as shown in the BIND rule in figure 4.4. A function call adds the exception result variable to the exception set so that the exceptions passed by a function are joined by the equation of the catch variable.

$$\textsc{Throw}: \frac{\mathtt{av}(l) \in \mathcal{X}}{\mathcal{R}, () \overset{\text{expr}}{\vdash} [\![\mathtt{throw}\ l]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{X}}$$

$$\textsc{Call}: \frac{foo_{exceptions} \in \mathcal{X}}{\ldots \overset{\text{expr}}{\vdash} [\![foo\ x_1\ \ldots\ x_n]\!] : \ldots, \mathcal{X}}$$

$$\textsc{Bind}: \frac{\begin{array}{l} foo_{exceptions} = \bigsqcup \mathcal{X} \in \mathcal{E} \\ \mathcal{R}, foo \overset{\text{expr}}{\vdash} [\![e]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{X}; \\ \ldots \end{array}}{\ldots \overset{\text{bind}}{\vdash} [\![foo\ a_1\ \ldots\ a_n = e]\!] : \ldots}$$

$$\textsc{Catch}: \frac{\begin{array}{l} \mathcal{R}, p \overset{\text{expr}}{\vdash} [\![b]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{X}' \\ l = \bigsqcup \mathcal{X}' \in \mathcal{E} \\ \mathcal{R}, p \overset{\text{expr}}{\vdash} [\![h]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{X} \end{array}}{\mathcal{R}, p \overset{\text{expr}}{\vdash} [\![\mathtt{try}\ \{b\}\ \mathtt{catch}(l)\ \{h\}]\!] : \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{X}}$$

Figure 4.4: Deriving equations for exception support

Chapter 5

Transformations

The GRIN compiler uses compilation by transformation [14]. A set of transformations, each small and easy to verify, is used to optimize and compile the code. Currenlty, only a subset of the transformations described in the Ph.D. thesis of Boquist [6] is implemented and no other transformations than those described by Boquist are implemented. Instead of copying the descriptions of those transformations from Boquist's thesis in this Chapter, we describe which transformations are implemented and refer to Boquist's thesis for the actual description.

5.1 Overview

Transformations on GRIN porgrams come in two flavors:

- simplifying transformations
- optimizing transformations.

The simplifying transformations remove syntactic sugar from GRIN. These transformations bring GRIN code into a very simple form, which is used as the input of the code generator. These transformations are mandatory: the code generator does not work without them.

The optimizing transformations are optional transformations. They change the representation of a program into a more efficient form. More efficient in this context means: smaller executables, less memory usage, shorter run time, or a combination of these factors.

Each transformation is implemented as a separate attribute grammer (AG) in a seperate file. Currently, most of the transformations have a fixed ordening, in particular simplifying transformations which work only on special forms of GRIN. The ordening of the transformations implemented within the GRIN compiler are shown in Figure 5.1.

5.2 Simplifying transformations

inline eval and apply The inlining of **apply** is exactly the same as described in Boquist's thesis. The inlining of **eval** uses a different version of the **eval** function

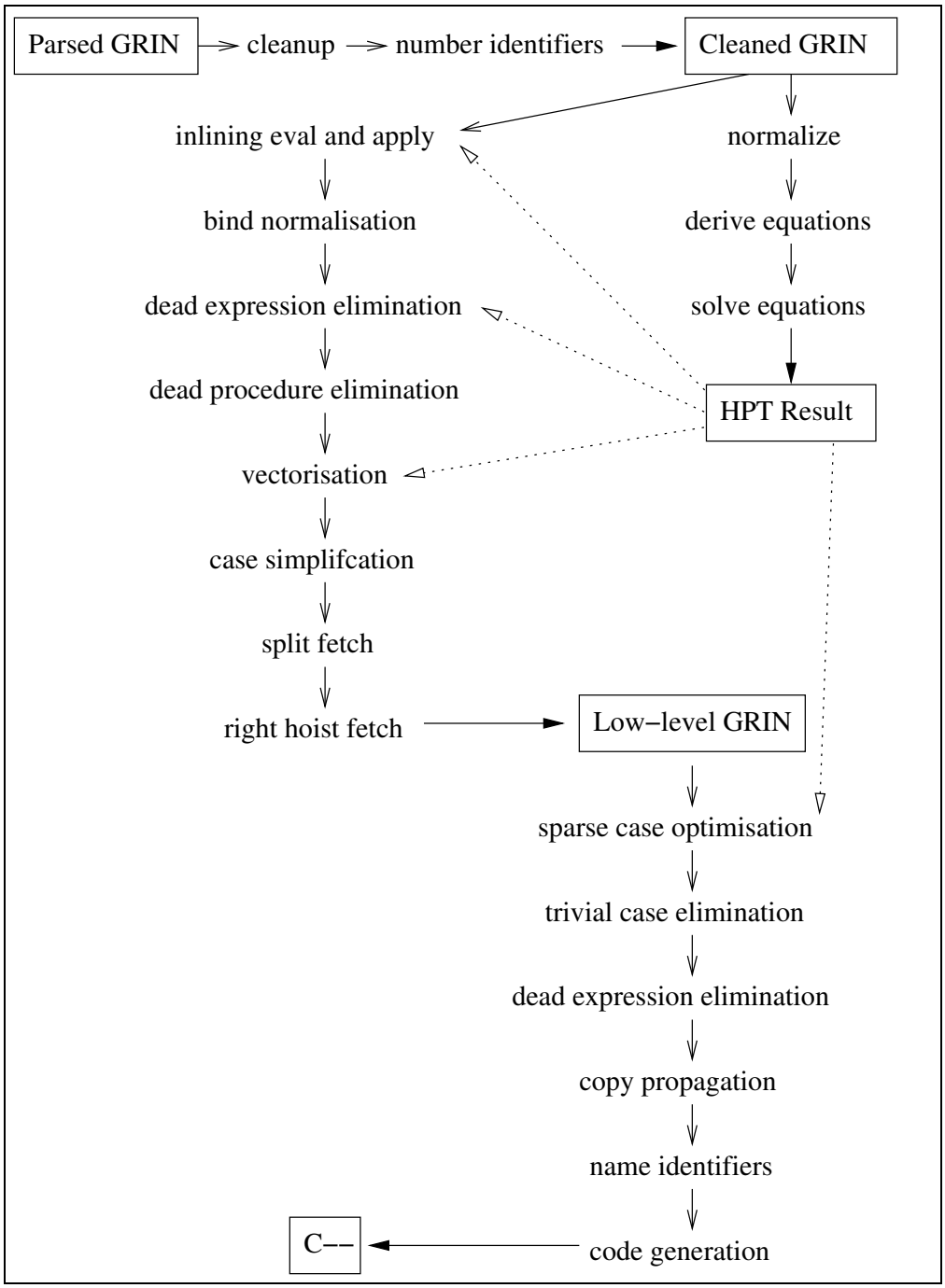

Figure 5.1: Overview of GRIN transformations

than the one uses by Boquist. The **eval** function described by Boquist contains an update statement after the case statement (page 85 of Boquist's thesis). Likely, this choice is made to be able to specialise the update statement later on (page 90 of Boquist's thesis). However update specialisation is not used in our implementation yet, and thus is not needed. Our version of the **eval** function is thus the original version, extended with exception support as show in Figure 4.2 on page 41.

Vectorisation and Case simplification The vectorisation and case simplification transformations have a fixed order: vectorisation changes node variables into a set of variables which introduces a variable to hold a tag. The case simplification then translates a case which scrutinizes such a set of variables, into a case which scrutinizes a tag variable. The transformations are easily combined into a single pass without losing readability.

Split and right hoist fetch operations The split fetch operations is implemented without taking into account the returning fetch operations described by Boquist. This situation does not occur in our implementation, as the transformations which lead to this situation are not implemented.

The right hoist fetch operations transformation is implemented, as described by Boquist, together with the split fetch operations transformation in a single pass.

5.3 Optimizing transformations

Copy propagation Copy propagation is used to remove the unit statements which are inserted by the normalisation of the code for HPT (Section 3.3.8). It is implemented as described by Boquist with one difference: the *right unit law* is not applied if it is preceded by a try-catch statement. This prevents the copy propagation to make the try-catch statement the last statement of a function body (Section 4.3).

Trivial case elimination and sparse case optimisation The trivial case elimination and sparse case optimisation are implemented as described by Boquist.

Dead code elimination Dead procedures elimination and the dead variables elimination is implemented. In our case the dead variable eliminations also includes the elimination of dead try-catch statements and the elimination of code after a throw statement. The HPT result records for each try-catch statement whether the handler will ever receive an exception. If this is not the case, the try-catch statement is replaced by the guarded statements.

The GRIN compiler uses a call-graph to find the dead procedures. As a bonus, the compiler can output this graph to a file, which can be used by graphviz[1] to generate a picture from like the one seen in figure 5.2.

[1] http://www.graphviz.org/

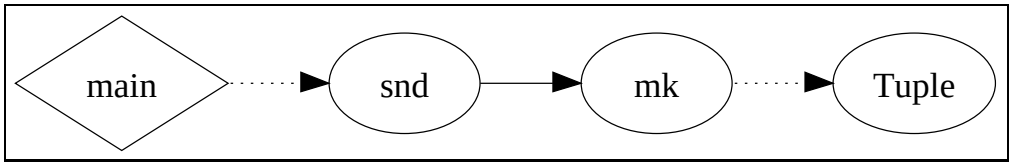

Figure 5.2: Call graph of running example

5.4 Miscellaneous transformations

cleanup The cleanup transformation is not described by Boquist. It is a 'glue' transformation which translates the EHC generated GRIN code into code which our GRIN compiler expects. The following constructs are translated:

- the generic eval and apply functions are dropped (grinc uses the eval- and applymap);
- a global variable is added for each CAF;
- the catch and throw foreign functions are rewritten to the catch and throw statements;
- P-nodes which represent partial applications missing zero arguments are rewritten to F-nodes (a bugfix of EHC).

All transformations done by this pass should be part of EHC and are only implemented in the GRIN compiler temporarily.

number identifiers and name identifiers Internally the GRIN compiler uses numbers. This allows the implementation to use constant time lookups of identifiers during the HPT analysis and while accessing the HPT result. The number identifiers transformation translates the names into numbers, the name identifiers transformation does the reverse.

bind normalisation The bind normalisation transformation is currently only needed for the eval and apply inlining which is the only transformation to replace a single statement with multiple statements.

Chapter 6

Code generation

The last phase of the GRIN compiler is code generation. Before the start of this phase the GRIN program is transformed by the simplifying transformations into *low-level* GRIN. Code generation is about tranforming this low-level GRIN is into C--, a portable assembly language designed by Peyton Jones et.al. [17, 18]. Once the C-- code is generated, the GRIN compiler is finished and the C-- compiler is left the delicate task of instruction selection and register allocation.

This chapter describes the low-level form of GRIN, the C-- language and the transformation of low-level GRIN to C--.

6.1 Low-level GRIN

After all simpifying transformations are applied to a GRIN program, the program is in low-level normal form. In this form each GRIN operation corresponds to a single, or a few, C-- instructions. Code generation is, because of this, a straightforward transformation.

In low-level GRIN, both **eval** and **apply** functions are inlined. Initially, these functions where partly built-in, partly represented by the eval and apply maps. The inlining of the calls to **eval** and **apply** make all function calls visible in the code; the eval and apply maps are not needed anymore.

All variables containing entire node values have been split into multiple variables, only variables containing pointers, tags or basic values remain. These values are big enough to be stored in a single register.

The **fetch** operations use offsets and load only a single field or tag. Each **fetch** operation on a field is also annotated with the tag of the node that the field is part of.

GRIN case statements scrutinise only tag values and bind no variables. The alternatives of a case expression are required be *exhaustive*, i.e. a case expression must have an alternative which matches the scrutinised value.

The running example translated into low-level GRIN is shown in figure 6.1. The long identifiers are the result of the various transformations which introduced new variables or splitted a single variable into multiple variables.

```
module "example"
  { main_caf ← store (#Fmain)
  }
  { Tuple x1 x2 = { unit (#CTuple x1 x2)
                  }
  ; snd t       = { fetch t 0; λt.27.38 →
                    case t.27.38 of
                      {#CTuple → { fetch_#CTuple t 1; λt.27.40.63 →
                                   fetch_#CTuple t 2; λt.27.41.62 →
                                   unit (t.27.38 t.27.40.63 t.27.41.62)
                                 }
                      ; #Fmk   → { fetch_#Fmk t 1; λt.27.40.66 →
                                   fetch_#Fmk t 2; λt.27.41.65 →
                                   mk t.27.40.66 t.27.41.65;
                                     λ(t.30.42 t.30.44 t.30.45) →
                                   update t (t.30.42 t.30.43 t.30.44 t.30.45);
                                     λ() →
                                   unit (t.30.42 t.30.44 t.30.45)
                                 }
                      }; λ(_ _ tn.49) →
                    fetch_#CInt tn.49 1; λ_69 →
                    unit (#CInt _69)
                  }
  ; mk f x4     = { fetch_#P1Tuple f 1; λf.35.58 →
                    Tuple f.35.58 x4
                  }
  ; main        = { store (#CInt 1); λs1 →
                    store (#P1Tuple s1); λs2 →
                    store (#CInt 2); λs3 →
                    store (#Fmk s2 s3); λs4 →
                    snd s4
                  }
  }
```

Figure 6.1: Low-level GRIN example

6.2 C--

A C-- compiler does what all compilers need to implement, but which is difficult to implement: it generates machine code. The C-- compiler does instruction selection, register allocation, instruction scheduling, and optimization of imperative code with loops.

C-- is designed to be a target for compilers, also called clients. C-- gives its client the freedom to implement a runtime model which suits the client best by providing a run-time interface to C--. This interface can be used to design run-time services for the client language, for example garbage collection or exception handling. A few highlights of C-- are:

- guaranteed tail calls;
- procedures which return multiple results in registers;
- stack cutting or unwinding by program code or through the run-time interface;
- linking data to parts of the program code, which can be used at run-time to pass information about the current point of execution to the run-time system.

C-- is designed to be portable. It hides architecture dependent details, such as the number of registers. But portability does not mean write once, run everywhere. Some properties of the architecture are exposed, for example the byte order and native word size. Encapsulating all details of a machine, like the Java Virtual Machine does, may, depending on the client language, impose a significant penalty in both space and time.

This section describes a subset of C--. A complete overview of C-- can be found on its website [1].

6.2.1 Syntax and semantics

The syntax of C-- is shown in the figures 6.2 and 6.3. It is a simplified version of C--. The symbol "?" in the figures denotates one or zero occurences, and "#" denotates zero or more occurences. The symbols "$*$" and "+" denotate respectively zero or more and one or more occurences, separated by commas.

A file with C-- source code is called a compilation unit. Such a compilation unit consists of a sequence of data sections, declarations, and procedures. Data sections consist of initialized or uninitialized data and labels to that data. Declarations can be type declarations, imported and exported names, constants, global registers, or a target declaration which describes the expected machine architecture of the compilation unit. All code is part of some parameterized procedure, which is defined at the top level of a compilation unit. Procedures have a fixed number of arguments and can return multiple values. Names of procedures and labels to data are visible through the whole compilation unit.

Variables in C-- are called registers. Registers have no address, and assigning to one cannot change the value of another. Registers may be local to a procedure or global to the program. Global registers are shared by all C-- procedures.

Procedures can define which calling convention must be used. Which conventions are available is up to the compiler, but C--defines two calling conventions which should always be available:

- the "C--" calling convention, which is the native calling convention. It allows tail calls and multiple return values in registers;
- the "C" calling conventions, which allows interaction with C.

The entry point of a C-- program is the procedure named *main*. This procedure is parameterized with two registers, returns one register and has the "C" calling convention.

Types, registers and memory

Registers in C-- have types. These types denote only the register size, like: *bits32 p* for a 32 bits register named p. Common sizes can be given names by the type definition statement. A boolean type, for example, can be defined by **typedef** *bool bits1*.

Loading and storing in memory is done by specifying a size and a memory location. For example: $bits8[p]$ loads one byte from the memory location whose address is stored in the variable p.

Expressions

Expressions consist of names, memory references, primitive operations, and literal values. Names are either names of registers or of link time constants. Link time constants are all labels, constant declarations, imported names, or function names.

Primitives are operations on registers and can be used to construct new values, like addition ($\%add$), multipy ($\%mul$), and modulo ($\%mod$) but also cover comparison of values, like equality ($\%eq$) and constants, like true ($\%true$). Primitives are free of side effects: they do not change memory, registers, or program flow.

Statements

A procedures consists of a sequence of statements which manipulate registers and memory. Assignments in C-- are *multiple assignments*; The values at the right-hand side and the addresses at the left-hand side are computed before the assignments are made. The types of each left-hand value must match the right hand value. An example statement which swaps a value in memory with a register:

$$bits32[p], x = x, bits32[p];$$

Control flow within procedures is influenced by if statements, switch statements and goto statements. An if statement needs an expression of type *bits1* and chooses the first body or second body depending on whether the bit is set or unset respectively. A switch statement is much like the if statement, but chooses a body based on the bit patterns defined in each alternative. When one of the patterns of an alternative matches that alternative is executed. Unlike in C, "fall through" between alternatives is not possible.

unit	::=	*toplevel#*	compilation unit
toplevel	::=	**section** "*data*" { *datum#* }	data section
	|	*decl*	
	|	*procedure*	
datum	::=	*name* :	label
	|	*type size*? *init*? ;	data
size	::=	[*expr*?]	
init	::=	{*expr*+}	initialization list
	|	*string*	initialization string
decl	::=	**target byteorder** (**little**|**big**) **memsize** *int* **pointersize** *int* **wordsize** *int* ;	target machine
	|	**import** *string* **as** *name* ;	import name
	|	**export** *name* **as** *string* ;	export name
	|	**const** *type*? *name* = *expr* ;	constant
	|	**typedef** *type name* ;	type definition
	|	*registers* ;	
registers	::=	*type name*+	variable declaration
procedure	::=	*name*(*formal*∗) { *bodyElem#* }	procedure defintion
formal	::=	*type name*	formal argument
bodyElem	::=	*decl*	
	|	*stackdecl*	
	|	*statement*	
stackdecl	::=	**stackdata** { *datum#* }	stack reservation
statement	::=	**if** *expr* { *bodyElem#* } **else** { *bodyElem#* }	if
	|	**switch** *expr* { *arm#* }	switch
	|	*lvalue*+ = *expr*+ ;	assignment
	|	*name*+ = *conv*? *name*(*expr*∗) *flow*∗ ;	procedure call
	|	*conv*? **jump** *name*(*expr*∗) ;	tail call
	|	**return** (*expr*∗) ;	return results
	|	*name* :	control label
	|	**goto** *name* ;	goto control label
	|	**continuation** *name*(*name*∗) :	continuation label
	|	**cut to** *expr*(*expr*∗) *flow*∗ ;	cut to continuation

Figure 6.2: Simplyfied syntax of C--, part 1

arm	::=	*case expr+ : { bodyElem# }*	switch alternative
lvalue	::=	*name*	register
	|	*type*[*expr*]	memory reference
flow	::=	**also cuts to** *name+*	cut to current activation
	|	**also aborts**	cut to deeper activation
	|	**never returns**	procedure never returns
conv	::=	**foreign** *string*	calling convention
expr	::=	*int*	integer literal
	|	*'char'*	character literal
	|	*name*	register
	|	*type*[*expr*]	memory reference
	|	(*expr*)	paranthesis
	|	*%name*(*expr**)	primitive operation
type	::=	**bits***N*	primitive type
	|	*name*	named type
string	::=	"*char**"	string

?	means zero or one
#	means zero or more
*	means zero or more, separated by commas (,)
+	means one or more, separated by commas (,)

Figure 6.3: Simplyfied syntax of C--, part 2

A control label can be used to change the control flow *within* procedures. A control label marks a point in the procedure. The label can be jumped to by the goto statement, but only if the goto statement and the label are enclosed by the same procedure.

A procedure call is not an expression, but a statement. It calls the named procedure with the arguments and saves the returned values in *result registers*. The result registers are the registers at the left hand side of the equation sign. The number of result registers must match the number of values returned by the call.

Calling a procedure creates an *activation*. The activation hold the values of the procedure's paramters and local registers. An activation dies when the function returns with a return statement or a cut to statement. A tail call replaces the current activation by an activation of the called procedure.

Continuations are labels which can be used to change the control flow *between* procedures. The label marks a point in the procedure and optionally marks registers to be used as parameters. The label name represents a value that marks this point and the current activation. This value is first class and from the native pointer size. The label is valid as long as the activation in which it is defined is alive. The control must never fall through a continuation label, they can only be accessed by a cut to statement.

A cut to statement jumps to the activation and location marked by a continuation label and passes the paramters along. All activations newer than the activation of the continuation are destroyed in the process.

The stackdata statement reserves uninitialized data on the stack. The reserved space is availible until the activation of the enclosing procedure dies.

Flow annotations

Procedure calls and cut to statements are annotated with flow information. This information informs the C-- compiler which continuations a procedure call or cut to statement might cut to. Continuations defined in the same procedure as the procedure call or cut to statement are recorded with the "also cut to" annotation; continuations in older activations are annotated with the "also aborts". Furhtermore, a procedure call might be annotated with "never returns" when the procedure call never returns by a return statement. If a procedure call or a cut to statement is not annotated, the "also aborts" annotation is assumed.

6.3 Transforming GRIN to C--

The code generator is quite simple. In fact, most of low-level GRIN maps one to one to C--. Once C-- is created the GRIN compiler is finished. Any optimisation on C-- code is done by the C-- compiler.

In this section we describe how the various GRIN values and constructs are represented in GRIN. The following aspects are discussed:

- representation of GRIN nodes and tags;
- representation of GRIN functions;

- representation of GRIN statements;
- flow annotations for exceptions.

The running GRIN example represented in C-- is shown in the figures 6.4 and 6.5.

The code generator must take care of a few machine dependent aspects of C--. Memory access and allocation is such an aspect. The GRIN compiler uses one size for each type of value; integers, pointers and tags are all of the native pointer size of the target machine. Currently this is only the x86 architecture. The size of a grin value can be calculated based on the pointer size and word size of the machine, on x86 architectures this is 32 bits and 8 bits respectively. Throughout this chapter we shall use the term word for the size of a grin value, rather than the machine word size. For the x86 architecture the grin word is equal to 4 machine words. All C-- code in this Section is given for the x86 architecture and refers to a word with the type *grWord*.

6.3.1 Exceptions

GRIN exceptions are modelled in C-- with the use of C-- continuations. Each try-catch statement introduces a continuation label to the start of the code of the exception handler. As long as the exception handler is active, the continuation label is stored in the *exception handler register*, named *eh*. A throw statement is expressed by rewinding the stack with a **cut to** statement to the label stored in the exception handler register.

C-- needs flow annotations on how the control changes if the stack is rewinded. Flow annotations are needed on **cut to** statements and function calls. Flow annotations are of the form **also cuts to** *name*, in which *name* is a continuation label which is defined in the same C-- procedure as the annotated statement. Thus an annotated statement must record which local defined exception handler it might activate. This is always the innermost try-catch statement. The tracking of the continuation label of the innermost try-catch statement is shown in Section 6.3.4 when discussing the transformation of GRIN statements. An example of C-- code with flow annotations is given in Appendix A.

6.3.2 Nodes and tags

GRIN makes two assumptions on the implementation of nodes and tags:

- the representation of each GRIN tag must be *unique*;
- extracting the tag and each field from a node value must be fast.

An unique tag allows a node in memory to be uniquely identified. This requirement is not neccessarily needed: tags must only be unique at each case statement[1]. However, program wide unique tags simplify accurate garbage collection.

The code generator assigns unique numbers to tags in the program, in the order of appearance. For readability, a constant is created with the same name as the tag which represents an unique number:

[1] this implies a certain uniqueness of tags stored in memory which is sufficient for garbage collection.

```
target memsize 8 byteorder little pointersize 32 wordsize 32;
typedef bits32 grWord;
grWord eh;
import "GC_malloc" as alloc;
export grin_main     as "grin_main";
const grWord Fmain   = 4;
const grWord Fmk     = 3;
const grWord P1Tuple = 2;
const grWord CTuple  = 1;
const grWord CInt    = 0;
section "data"{ GlobalNodes :
                  main_caf : grWord[5]{ Fmain, 0, 0, 0, 0 };
                  GlobalNodesEnd :
                }
section "data"{ TAGInfoTable :
                  grWord[1]{ 1 };
                  grWord[1]{ 2 };
                  grWord[1]{ 1 };
                  grWord[1]{ 2 };
                  grWord[1]{ 0 };
                }
grin_main (){
  grWord _, result;
  _, result = main ();
  return (result);
}
main (){
  grWord _;
  grWord s1;
  s1 = foreign "C" alloc (20);
  grWord[s1], grWord[%add (s1, 4)] = CInt, 1;
  grWord s2;
  s2 = foreign "C" alloc (20);
  grWord[s2], grWord[%add (s2, 4)] = P1Tuple, s1;
  grWord s3;
  s3 = foreign "C" alloc (20);
  grWord[s3], grWord[%add (s3, 4)] = CInt, 2;
  grWord s4;
  s4 = foreign "C" alloc (20);
  grWord[s4], grWord[%add (s4, 4)], grWord[%add (s4, 8)] = Fmk, s2, s3;
  jump snd (s4);
}
```

Figure 6.4: Example C-- program, part 1

```
Tuple (grWord x1, grWord x2){
  grWord _;
  return (CTuple, x1, x2);
}
snd (grWord t){
  grWord _;
  grWord t.27.38;
  t.27.38 = grWord[t];
  grWord tn.49;
  switch t.27.38{
    case CTuple : {
      grWord t.27.40.63;
      t.27.40.63 = grWord[%add (t, 4)];
      grWord t.27.41.62;
      t.27.41.62 = grWord[%add (t, 8)];
      _, _, tn.49 = t.27.38, t.27.40.63, t.27.41.62;
    }
    case Fmk : {
      grWord t.27.40.66;
      t.27.40.66 = grWord[%add (t, 4)];
      grWord t.27.41.65;
      t.27.41.65 = grWord[%add (t, 8)];
      grWord t.30.42, t.30.44, t.30.45;
      t.30.42, t.30.44, t.30.45 = mk (t.27.40.66, t.27.41.65);
      grWord[t], grWord[%add (t, 4)], grWord[%add (t, 8)] = t.30.42, t.30.44, t.30.45;
      _, _, tn.49 = t.30.42, t.30.44, t.30.45;
    }
  }
  grWord _69;
  _69 = grWord[%add (tn.49, 4)];
  return (CInt, _69);
}
mk (grWord f, grWord x3){
  grWord _;
  grWord f.35.58;
  f.35.58 = grWord[%add (f, 4)];
  jump Tuple (f.35.58, x3);
}
```

Figure 6.5: Example C-- program, part 2

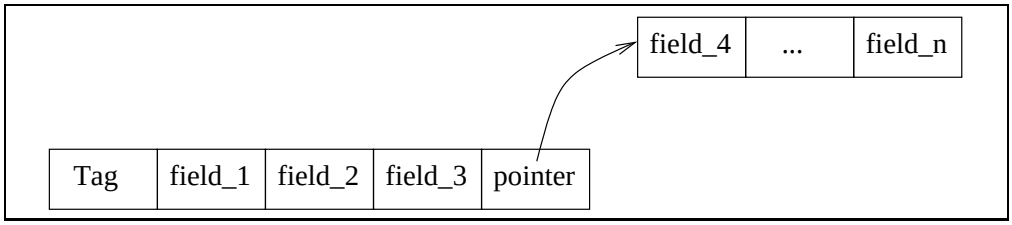

Figure 6.6: Node layout

$$[\![\#Tag]\!] \mapsto \textbf{const}\ Tag = uniq$$

The ordening of the tag numbers does influences the code quality: a case statement can be implemented efficiently when the tags listed in the alternatives are numbered without gaps. For example a case with the tags 1, 2, 3, and 4 can be implemented more efficiently than a case with the tags 1, 7, 11, and 14. A possible optimisation is to analyse the case statements and choose a numbering so that the numbers of the tags within each case are as close to each other as possible.

To extract the tag or a field from a node is needed for efficient implementation of the fetch and update statements. A typical Haskell program will translate to a GRIN program with many fetch and update statements. Thus it is very important that these statements can be implemented to run as fast as possible.

To implement the second requirement, the layout of nodes in memory is simple (Figure 6.6). A node in memory is always 5 words big, this is called the *base part* of the node. The tag and up to 3 fields are stored in the base part. The last word of the base part is reserved for a pointer which, when the node has more than 3 fields, points to the *extended part* of the node. The extended part of a node consist of just enough memory to store the additional fields of a node with more than 3 fields. This node layout supports updating a small node with a large node while keeping the original pointer to the node unchanged.

The node layout wastes precious memory: nodes with 4 fields are stored inefficiently. But the layout is choosen to keep the implementation of the GRIN compiler simple. A more conservative node layout is possible, but needs transformations not yet implemented in the GRIN compiler.

Global variables

Global variables and the nodes they point to are represented by labels and values of a data section respectively. This data section is conform the node layout. The global variables are translated as follows:

$$\left[\!\!\left[\begin{array}{l} x_1 \leftarrow \mathbf{store}\,(\#\mathit{Tag1}\ v_1 \ldots v_m); \\ \ldots \\ x_n \leftarrow \mathbf{store}\,(\#\mathit{Tag2}\ y_1 \ldots y_p); \end{array} \right]\!\!\right] \mapsto \begin{array}{l} \mathbf{section}\ \texttt{"data"}\{ \\ \quad x_1 \quad\ \ : \mathit{grWord}[5]\{\,\mathit{Tag1}, v_1, \ldots v_3, x_1.\mathit{big}\,\} \\ \quad x_1.\mathit{big} : \mathit{grWord}[]\ \{\,v_4, \ldots v_m\,\} \\ \quad \ldots \\ \quad x_n \quad\ \ : \mathit{grWord}[5]\{\,\mathit{Tag2}, y_1, \ldots y_p, 0, \ldots\,\} \\ \} \end{array}$$

with $m \geqslant 4$, $p < 4$

The node pointed to by x_1 has more than 3 fields and is split in two in the data section. The node pointed to by x_n is smaller than 4 fields, and is padded with zeros to fill the 5 words value.

6.3.3 Functions

GRIN functions can have multiple statements on the return spine. For example, a GRIN function which returns a Haskell value of the type *Maybe Int* can return (#*CNothing*) or (#*CJust p*). These values are represented in low-level GRIN with one or two values, respectively. Thus, a GRIN function can return a variable number of values.

A C-- procedure cannot vary in the numbers of returned values: the call side of a C-- procedure must match the number of values returned by that procedure. To create a mapping of GRIN function to C-- procedures we must ensure that a GRIN function returns a fixed number of values. To this end, the code generator maintains a map from function name to the maximum number of values returned. This map is used by the statements at the return spine to pad the number of returned values.

Each GRIN function is translated into a C-- procedure and a variable definition. This variable has the same purpose as the wildcard found in GRIN patterns and is denoted by an underscore (_).

$$[\![\mathit{foo}\ x_1 \ldots x_n = e]\!] \mapsto \mathit{foo}\,(x_1, \ldots, x_n)\{\,\mathit{grWord}\ \_; [\![e]\!]\ \text{``}\ ()\,\}$$

6.3.4 Statements

GRIN statements have two representations in C--, one when the statement occurs on the return spine, and one when the statement does not. The translation rules are described by:

$$[\![e]\!]\ n\ p \mapsto c$$

This rule will translate the GRIN statement e to the C-- code c. Patterns in GRIN are translated to a list of target registers, which is represented above by the variable p. The target registers represent in which registers the result of the statement must be stored. When () is passed as register pattern, the statement is on the return spine. The different forms of register patterns are generated by the GRIN sequence operator.

The variable n in the translation rules represent an active locally defined try-catch statement. This variable can be either empty (''), when no try-catch statement is active in the current GRIN function or hold the name of the continuation label which should be added as annotation to a **cut to** statement or function call. All rules ignore this value, except those who tranform a **throw** statement or function call.

Sequence

The GRIN sequence operator translates a GRIN pattern into target registers, and generates C-- code which defines these patterns. Tag literals and wildcards in a GRIN pattern are translated to the C-- register _, which is defined at the top of each procedure. An empty GRIN pattern, denoted by (), is translated into an empty list of target registers.

$$
\begin{aligned}
[\![e; \lambda x \rightarrow b]\!]\ n\ p &\mapsto grWord\ x; \\
&\quad\ [\![e]\!]\ n\ x \\
&\quad\ [\![b]\!]\ n\ p \\
\\
[\![e; \lambda() \rightarrow b]\!]\ n\ p &\mapsto [\![e]\!]\ n\ \{\} \\
&\quad\ [\![b]\!]\ n\ p \\
\\
[\![e; \lambda(Tag\ x_1 \ldots x_n) \rightarrow b]\!]\ n\ p &\mapsto grWord\ x_1, \ldots, x_n; \\
&\quad\ [\![e]\!]\ n\ \{\_, x_1, ..., x_n\} \\
&\quad\ [\![b]\!]\ n\ p \\
\\
[\![e; \lambda(t\ x_1 \ldots x_n) \rightarrow b]\!]\ n\ p &\mapsto grWord\ t, x_1, \ldots, x_n; \\
&\quad\ [\![e]\!]\ n\ \{t, x_1, ..., x_n\} \\
&\quad\ [\![b]\!]\ n\ p
\end{aligned}
$$

Unit

The unit statement translates into an assignment of registers, constants, and literals to the target registers. When the statement occurs on the return spine, a return statement is genereted instead.

$$
\begin{aligned}
[\![\mathbf{unit}\,(Tag\ x_1 \ldots x_n);]\!]\ n\ \{y0, ..., yn\} &\mapsto y_0, \ldots, y_n = Tag, x_1, \ldots, x_n; \\
\\
[\![\mathbf{unit}\ x;]\!]\ n\ y &\mapsto y = x; \\
\\
[\![\mathbf{unit}\,(Tag\ x_1 \ldots x_n);]\!]\ n\ () &\mapsto \mathbf{return}\,(Tag, x_1, \ldots, x_n); \\
\\
[\![\mathbf{unit}\ x;]\!]\ n\ () &\mapsto \mathbf{return}\,(x);
\end{aligned}
$$

Update

The update statement must update the node in memory. A node with more than 3 fields is split in a base part and a extended part. The extended part is allocated by the update statement, if it is needed. The base part of a node holds upto 3 fields, so the amount of memory needed is $fieldcount - 3$ words, where $fieldcount$ is the number of fields of the node.

The assignments to memory are emitted as shown below. The variable _ is used to hold the newly allocated extended part of the node while the update is in progress.

$$
\begin{array}{lll}
[\![\textbf{update}\ x\,(Tag, x_1, \ldots x_n)]\!]\ n\ \{\} & \mapsto & \_ = alloc\,(4*(n-3)); \\
 & & grWord[x], \\
 & & grWord[\%add\,(x,4)], \\
 & & \ldots, \\
 & & grWord[\%add\,(x,16)] \qquad\qquad = Tag, x_1, \ldots, \_; \\
 & & grWord[\_], \\
 & & grWord[\%add\,(\_,4)], \\
 & & \ldots, \\
 & & grWord[\%add\,(\_,(n-4)*4)] = x_4, x5, \ldots, x_n;
\end{array}
$$

The above transformation is for the update of a node with more than 3 fields. If a node contains less fields, the first and the last of the three statements above are not emitted.

Store

A store statement is similar to the update statement. The only diffenrence is that the store statement does also allocates the base part. Store is implemented as an allocation of 5 words, the base part, after which the same code as the update statement is emitted.

$$
[\![\textbf{store}\ node]\!]\ n\ x \quad \mapsto \quad x = alloc\,(20); [\![\textbf{update}\ x\ node]\!]\ n\ \{\}
$$

Fetch

A fetch statement loads the tag or a field of a node from memory. Depending on the offset given to fetch, a field from the base part, or from the extended part is loaded. To load a field from the extended part, the fetch statement must first load the pointer to the extended part from the base part, and then load the field from memory.

$$
\begin{array}{lll}
[\![fetch_{\#Tag}\ p\ small]\!]\ n\ f & \mapsto & f = grWord[\%add\,(p, small*4)]; \\
[\![fetch_{\#Tag}\ p\ small]\!]\ n\ () & \mapsto & \textbf{return}\,(grWord[\%add\,(p, small*4)]); \\
[\![fetch_{\#Tag}\ p\ big]\!]\ n\ f & \mapsto & \_ = grWord[\%add\,(p,16)] \\
 & & f = grWord[\%add\,(\_,(big-4)*4)];
\end{array}
$$

$$[\![fetch_{\#Tag}\ p\ big]\!]\ n\ () \quad \mapsto \quad \begin{array}{l} \_ = grWord[\%add\,(p, 16)] \\ \textbf{return}\,(grWord[\%add\,(\_,(big - 4) * 4)]); \end{array}$$

where $0 < small \leqslant 3$ and $big > 3$

Case

The GRIN case statment in low-level form is semantically equivalent to a C-- switch statement. The transformation is thus only a syntactical rewrite.

$$\left[\!\!\left[\begin{array}{l} \textbf{case}\ x\ \textbf{of} \\ \quad T_1 \rightarrow e_1 \\ \quad \ldots \\ \quad T_N \rightarrow e_n \end{array} \right]\!\!\right] n\ p \quad \mapsto \quad \begin{array}{l} \textbf{switch}\,(x)\{ \\ \quad \textbf{case}\ T_1 : \{\ [\![e_1]\!]\ n\ p\ \} \\ \quad \ldots \\ \quad \textbf{case}\ T_N : \{\ [\![e_n]\!]\ n\ p\ \} \\ \} \end{array}$$

Try-catch

The try-catch statement is the most complex statement. The code generator emits code which does the following:

1. save the current exception handler;
2. install the new exception handler;
3. execute guarded code;
4. restore the saved exception handler.

In addition, the code generator emits code for the new exception handler. This code starts with a continuation label. The scope of a label in C-- is the procedure body, which gives freedom in the location of the code for the exception handler. The transformation given here emits the exception handler directly after the code which restores the saved exception handler. The actual implementation collect all exception handlers and add those at the end of the procedure body. This limits the number of jumps if no exeception is thrown.

Saving the current exception handler and installing the new one is done with a single C-- statement. The handler is restored after the guarded statements, and before executing the first statement of the exception handler. To prevent the execution of the handler when no exception occurs, a jump is emitted before the exception handler which skips the exception handler.

The name of the continuation label of the exception handler is passed to the guarded statements which use this name to emit flow annotations.

$$\begin{array}{lll} [\![\textbf{try}\{\,b\,\}\textbf{catch}\,(e)\{\,h\,\}]\!]\ n\ p & \mapsto & \textbf{stackdata}\{\,prev.eh.e : grWord;\,\} \\ & & grWord[prev.eh.e], eh = eh, handler.e; \\ & & [\![b]\!]\ \text{`}handler.e\text{'}\ p \\ & & eh = grWord[prev.eh.e]; \end{array}$$

goto *after.e*;
grWord e;
continuation *handler.e* (*e*) :
eh = *GrWord*[*prev.eh.e*];
⟦*h*⟧ *n p*
after.e:

Throw

The throw statement cuts the stack to the activation of the active exception handler. A continuation label to the active exception handler is stored in the global register *eh*. The **cut to** statement cuts to this register, passing the exception pointer along. When the active exception handler is defined locally, the continuation label of this exception handler is added in the flow annotation **also cuts to**.

$$\begin{array}{rcl} [\![\textbf{throw } p]\!] \text{ `'} \ p & \mapsto & \textbf{cut to } eh\,(p) \\ [\![\textbf{throw } p]\!] \text{ `}foo\text{'} \ p & \mapsto & \textbf{cut to } eh\,(p) \textbf{ also cuts to } foo \end{array}$$

Function call

A GRIN function call translates directly into a C-- function call. If the active exception handler is defined locally, its name is added in the flow annotation. When the function call occurs on the return spine, a tail call is emitted.

$$\begin{array}{rcl} [\![foo\ x_1 \ldots x_n]\!] \text{ `'}\{y0, ..., y_n\} & \mapsto & y_0, \ldots, y_n = foo\,(x_1, \ldots, x_n); \\ [\![foo\ x_1 \ldots x_n]\!] \text{ `}bar\text{'} \ \{y0, ..., y_n\} & \mapsto & y_0, \ldots, y_n = foo\,(x_1, \ldots, x_n) \textbf{ also cuts to } bar; \end{array}$$

$$[\![foo\ x_1 \ldots x_n]\!] \text{ `'} \ () \quad \mapsto \quad \textbf{jump } foo\,(x_1, \ldots, x_n);$$

Foreign function call

When a foreign function call calls a primitive operation, emit the code for this primitive. Otherwise, import the function name and emit a call to it with the "C" calling convention.

$$\begin{array}{rcl} [\![\textbf{ffi } primFoo\ x_1 \ldots x_n]\!] \ n \ p & \mapsto & emitPrimitive\,(p, primFoo, x_1, \ldots, x_n) \\ [\![\textbf{ffi } foo\ x_1 \ldots x_n]\!] \ n \ y & \mapsto & \textbf{import } foo; \\ & & foreign \texttt{ "C" } foo\,(x_1, \ldots, x_n) \end{array}$$

Chapter 7

Conclusions and future work

7.1 Conclusions

On of the goals of this thesis was the implementation of a back-end for EHC based on GRIN. This goal is met for a big subset of the EH language. The language constructs which cannot be compiled yet are the **update** statement outside the **eval** function in GRIN, which is merely a practical problem with HPT, and support for (extensible) records in GRIN. Extensible records support was not included in the thesis goals and is thus not implemented. Non extensible records are not supported because EHC uses the same GRIN constructs as with the extensible records, which makes them, from the GRIN perspective, equal to extensible records.

The current implementation is still in the 'toy' phase: it does not generate efficient code, because only a few optimizing transformations are implemented, and an accurate garbage collector is absent, which makes big programs choke on the amount of memory they need. However, the current implementation is a good basis to create a full featured GRIN compiler.

The research goals of this thesis are the description of HPT and the extension of GRIN with support for exceptions. We have described the working of HPT and have shown how HPT can be adapted, without difficulty, to support exceptions in GRIN. Furthermore, we have shown that with the added exception support, the imprecise exceptions extension of Haskell can be expressed in GRIN.

7.1.1 Implementation experience

The complete implementation, except the solving of the equations of HPT, is created with the AG system uuagc [5]. The various transformations could be expressed quickly in AG.

There is a downsite to the used AG system. The used AG system cannot expresses references to other nodes in the tree, a feature which is found in reference AG systems, like JastAdd [4]. Since we cannot reference other nodes, we must collect the contextual information needed by the transformations ourself. Contextual information might be absent, so we must pack the information in a datatype which can represent the existence or absence of the expected context, for example usening Haskell's *Maybe* datatype.

Equations based on these attributes must be aware that the needed context is absent, and, the more context is needed, the more unreadable those equations become.

Another downside of the AG system is the constant rebuilding of the AST. Some transformations work very local in the AST and subtrees of the AST are guarenteed not to change. The current implementation of the AG system is not aware of the unchanged subtrees and will simply deconstruct and rebuild the whole AST, even if this result in an identity transformation.

7.2 Future work

To create a full featured EHC back-end based on GRIN, more research and implementation work is necessary. This section describes possibilities for further research and lists several remaining implementation issues.

7.2.1 Thread support and asynchrone exceptions

A challenging part of GRIN is the modelling of thread support and asynchrone exceptions. Especially the question if locking and the communication between threads can be analysed and optimized with GRIN.

7.2.2 Separate compilation

One problem of whole program compilation is the compile time; a small change in a function means recompilation of the complete program. To prevent complete recompilation of the program, the front-end can use separate compilation and emit GRIN code for each Haskell module seperately. The GRIN compiler is invoked with a set GRIN modules and joins those modules together to create the complete GRIN program.

While this is better than complete recompilation, the GRIN compiler still recompiles the whole program for each change. A valid question would be if the GRIN compiler can cache some information to speed-up the recompilation process.

HPT is a good candidate for caching. While HPT gives only a valid estimation if the whole program is available, it is possible to run HPT on a part of the program. The result from such a run is not complete and cannot be used for transformations, but it can be used as the start of HPT on a complete program, saving some iterations needed to reach a fixpoint.

One step further would be to make the HPT analyse the distribution of unknown values. The HPT can be modified to have a notion of pointers to external heap locations, and unknown tags. Some help from the front-end is needed to analyse the evaluation of an unknown closure. The type information availible by the front-end is a good source to approximate the evaluation of unknown closures.

HPT results based on a module are worse than an analysis on a complete program, but some transformations, like the vectorisation transformation, can already be applied to part of a GRIN module. Results of this modified HPT can also be used as a starting point for HPT on a complete program, provided that all information based on an external heap location can be removed from the result of HPT on a module.

7.2.3 GRIN vs STG-machine

It would be interesting to do some study on the differences and similarities between GRIN and the STG-machine [13] as used by GHC.

One difference, the tags in GRIN vs the tagless STG-machine, is not so big as it might seem. Tags in GRIN are identifiers of nodes. Currently, we use an integer as the runtime representation of a tags, but using an unique pointer as as the representation of a tag is also possible. Such a tag can point to a record containing pointers to code for evaluation, garbage collection, etc. This comes very close to the tagless part of the STG-machine.

One possible use of pointers as tags would be to prevent very big eval case statements. If HPT gives many possible nodes for an eval call, the case statements which are created by inlining the eval calls are huge. When using pointers as tags, we have the option to use the tag as a pointer and jump to the code for evaluation instead of inserting a very big case statement, while for small case statements a jump table can still be used, although it is a bit more inefficient to calculate the jump offset based on pointers instead of integers.

7.2.4 Extensible records

EHC supports extensible records as described by Simon Peyton Jones [11]. For this purpose EHC emits some special GRIN constructs. This section gives a start on how these GRIN constructs can be compiled. Throughout this section we use the construct **case** $a > b$ **of** ... as a shorthand for:

ffi *primGtInt a b*;
$\lambda c \rightarrow$ **case** c **of** ...

Extensible records as described by Simon Peyton Jones have the property that all possible sizes of the records are finite and known at compile time. Thus, it is possible to give each record size an unqiue tag in GRIN. E.g., a record with zero elements gets the name #*R0*, a record with one element #*R1*, and so on. At runtime these tags are represented by a continuous interval of integers. E.g., #*R0* is assigned the value n, #*R1* the value $n + 1$, and so on. If the size of a record is needed at runtime this can be calculate by substracting the number assigned to tag #*R0* from the number assigned to the tag representing the record.

To express a selection from a record in GRIN, EHC uses a special case alternative:

case r **of**
  $(\#R\_\ n \mid f = o) \rightarrow \ldots$

This construct selects a field at offset o from record r and binds it to f. The variable n represents a record of one size less than the record r, e.g. n contains all the fields of r minus f. The tag #$R\_$ in the alternative pattern indicates that the size of the record, and thus the tag of the node, is not known.

Once the GRIN compiler starts, the whole program is available, and since all record sizes are known at compile time all possible record tags are available in the program.

HPT analysis already collects which record tags, and thus which record sizes, might be bound to a record selection statement. When the possible tags are estimated, the record selection statement can be rewritten to a normal case statement. The variable name of the selected field is found by a binary search based on the offset. Offsets are basic (unboxed) integers. The offset 1 represents the offset of the first field:

$$
\begin{array}{l}
\textbf{case}\ r\ \textbf{of} \\
\quad \{(\#R2\ x_1\ x_2) \qquad \rightarrow \textbf{case}\ o > 1\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad \#CTrue \rightarrow \textbf{unit}\,(\#R1\ x_2\ x_1) \\
\qquad\qquad\qquad\qquad\quad \#CFalse \rightarrow \textbf{unit}\,(\#R1\ x_2\ x_1) \\
\quad ;(\#R4\ y_1\ y_2\ y_3\ y_4) \rightarrow \textbf{case}\ o > 2\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad \#CTrue \rightarrow \textbf{case}\ o > 3\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \#CTrue \rightarrow \textbf{unit}\,(\#R3\ y_4\ y_1\ y_2\ y_3) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \#CFalse \rightarrow \textbf{unit}\,(\#R3\ y_3\ y_1\ y_2\ y_4) \\
\qquad\qquad\qquad\qquad\quad \#CFalse \rightarrow \textbf{case}\ o > 1\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \#CTrue \rightarrow \textbf{unit}\,(\#R3\ y_2\ y_1\ y_3\ y_4) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \#CFalse \rightarrow \textbf{unit}\,(\#R3\ y_1\ y_2\ y_3\ y_4) \\
\quad \};\lambda(v_0, f, v_1, v_2, v_3) \rightarrow \textbf{unit}\,(v_0, v_1, v_2, v_3); \lambda n \rightarrow \\
\ldots
\end{array}
$$

The first field of the node returned by the case statement is the selected field. This makes the node one field bigger than it should be according to its tag. But as long as we don't analyse the code again this is safe. Immediately after the case statement we destruct the returned node. f is bound to the first field and n is bound to the tag and other fields, which represent the record without the selected field.

HPT can also be used to collect a set of possible offsets for each variable holding a record offset. To discriminate offset constants from other constants, the front-end must annotate the offset constants. If all offset constants are known, HPT can find a set of integers for offset variable instead of "$\{B\}$", which it currently does.

In the special case that only a single record size is found, and this record is stored in memory we might perform much better by emitting a fetch statement with the offset of the record field to select as fetch offset, e.g. **fetch** $p\ o$, in which p is a pointer to record r. However, in most cases a thunk of a function which returns a record is stored in the same memory location as the record itself, which makes this situation hard to detect.

A unsolved problem however is how to estimate the result of f. This variable can hold any value stored in the possible records. To get a good estimation, the relation between the record sizes and the offsets must be taken into account. E.g., HPT must analyse which tags and offsets combinations occur in the selection statement.

To extend records, EHC uses the statement **unit** $(r \mid o{+}{=} f)$. This extends a record r with field f at offset o. Each field with an offset greater than or equal to o is shifted to the right. The record extension statement can be rewritten in the same way as the record selection. After the possible record tags are estimated by HPT, a record extension statement can be rewritten to a normal case statement:

$$
\begin{array}{l}
\textbf{case}\ r\ \textbf{of} \\
\quad \{(\#R1\ x_1) \qquad \rightarrow \textbf{case}\ o > 1\ \textbf{of} \\
\qquad\qquad\qquad\quad \#CTrue \rightarrow \textbf{unit}\,(\#R2\ x_1\ f)
\end{array}
$$

```
                    #CFalse → unit (#R3 f x1)
;(#R3 y1 y2 y3) → case o > 2 of
                    #CTrue  → case o > 3 of
                                #CTrue  → unit (#R4 y1 y2 y3 f )
                                #CFalse → unit (#R4 y1 y2 f  y3)
                    #CFalse → case o > 1 of
                                #CTrue  → unit (#R4 y1 f  y2 y3)
                                #CFalse → unit (#R4 f  y1 y2 y3)
}
```

As with the selection satement, the extension statement HPT makes a bad estimation on the result of this statment. Again HPT must analyse which tags and offsets can occur at the same time.

7.2.5 Update specialisation

The update statement has currently two problems:

- the update statement cannot differ in the layout of the node in memory;
- the update statement might write uninitialized data to memory.

The current implementation uses a single layout of nodes in memory. This layout is shown in Figure 6.6 on page 59. A problem with this layout is explained by the following example: a node with 4 fields is stored in memory. Following the current layout, this node is split into a base part, which hold the tag, 3 fields, and a pointer to the extended part, and an extended part, which holds the 4th field. A better layout would be to store the 4th field also in the base part, but this is only possible if the update, store, and fetch statements know that the node has 4 fields.

The second problem is shown in the following example: we define a function *upto*, which returns either a node with 2 fields or a node without fields, and a closure to this function. The code responsable for the evaluation of this closure is as follows:

```
fetch p; λ(nt n1 n2) →
case nt of
  {#CCons → unit (nt n1 n2)
  ;#CNil  → unit (nt)
  ;#Fupto → upto n1 n2; λ(t x1 x2) →
            update p (t x1 x2); λ() →
            unit (t x1 x2)
  }
```

The update statement in this code always writes x_1 and x_2 to memory, while these variables might be uninitialized. To prevent writing of unitinialized variables to memory, the update statement must know the size of the node.

Boquist describes how fetch and update can be annotated with the size, or actually the tag, of the stored node.[1] The annotation of the fetch statement is already implemented

[1] the store statement has always a tag constant of the node it stores.

by the current implementation (as part of the "right hoist fetch transformation"). To annotate, or specialize, the update statement, Boquist uses a different **eval** implementation. An example of this version of **eval**, extended with our exception support, is as follows:

```
fetch p; λn →
try{
  case n of
    {(#CCons x1 x2) → unit n
    ;(#CNil)        → unit n
    ;(#Fupto a1 a2) → upto a1 a2
    ;(#Fthrow e)    → throw e
    }
}catch(e){
    update p(#Fthrow e); λ() →
    throw e
}; λr →
  update p r; λ() →
  unit r
```

Compared to the version used in our implemenation (shown in Figure 4.2 on page 41) this version of eval is less efficient since it might update a node with itself. The update statement which updates the thunk with its result is put after the try-catch statement. This is possible because an update statement never throws an exception[2] and the last update statement of eval is never reached if the exception handler is executed.

This version is more suitable for specializing the update statement using Boquists approach of right hoisting the update statement until the tag of the node in the update statement is known. However, the HPT can also be used to insert a case statement which will descirbes the tag:

```
fetch p; λn →
  case n of
    {(#CCons x1 x2) → unit n
    ;(#CNil)        → unit n
    ;(#Fupto a1 a2) → {try{
                          upto a1 a2; λr →
                          case r of
                            {(#CCons x1 x2) → update#CCons p r
                            ;(#CNil)        → update#CNil p r
                            }λ() →
                          unit r
                       }
                      }
    ;(#Fthrow e)    → throw e
    }
```

[2] it might throw an out of memory exception, but that exception is treated as asynchrone.

This way, the unnecessary updates are avoided. Which version leads to the fastest running time is an open question.

7.2.6 Haskell's builtin monad support

The IO and the ST monad in Haskell do not necessarily need special support in GRIN. To keep the sequential composition of these monads in tact in the presents of optimisations, a sequence number (e.g. an integer) can be used as representation of the state of the monad. Before each site effect, encoded by an **ffi** statement, a sequence number must first be increased.

With this trick, code moving transformations must still be aware not to break the ordening of ffi statements and the code that updates the sequence number. To make this tracking easier to implement, we can use a special *marker* value instead of sequence number and a *touch* statement instead of the increment of the sequence number. A touch statement takes a marker variable and returns a changed marker value. Instead the hard to detect code block of the increment of a sequence number, we have now a single touch statement. Any code moving transformation must not break the ordening of ffi statements and touch statements.

The marker value and the touch statements are only used to make the sequencing of the IO and ST monad explicit in GRIN. When all code moving transformations are finished, any reference to a marker value and the touch statements can be removed from the GRIN program.

7.2.7 To do list

We list here some tasks which must be implemented sooner or later.

- define a correlation between the key and the value of the eval- and applymap, such that those maps become obsolete;
- add support for blackholing in **eval**;
- HPT support for update statements anywhere in the code, or at least support for self referencing values;
- add sharing analysis to HPT. The current implementation has already parts of HPT modified to support the sharing analysis;
- use dependency information of the equations in HPT to reach a fixpoint faster using a worklist algorithm;
- implement more transformations. Good starters are: Unboxing, constants propagation, late inlining;
- allow experimentations of the ordening of (optimizing) transformations from the command line;
- allow ffi statements to return boxed values;
- optimize tag number assignment;
- implement accurate garbage collector.

Bibliography

[1] C--. URL http://www.cminusminus.org/.

[2] Essential haskell compiler. URL http://www.cs.uu.nl/wiki/Ehc.

[3] Glasgow haskell compiler. URL http://www.haskell.org/ghc/.

[4] Jastadd - a java-based, aspect oriented compiler compiler system. URL http://jastadd.cs.lth.se/web/.

[5] uuag - utrecht university attribute grammar system. URL http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem.

[6] Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology and Goteborg University, 1999. URL http://www.cs.chalmers.se/~boquist/phd/index.html.

[7] Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Implementation of Functional Languages*, pages 58–84, 1996. URL http://citeseer.ist.psu.edu/boquist96grin.html.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. ISSN 0164-0925. URL http://doi.acm.org/10.1145/115372.115320.

[9] Chris Hankin Flemming Nielson, Hanne R. Nielson. *Principles of Program Analysis*. Springer-Verlag, 1999.

[10] T. Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6): 58–69, June 1984. URL http://citeseer.ist.psu.edu/johnsson84efficient.html.

[11] M. Jones and S. Jones. Lightweight extensible records for haskell, 1999. URL citeseer.ist.psu.edu/jones99lightweight.html.

[12] S. L. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A non-strict, purely functional language. Technical report, feb 1999. URL http://www.haskell.org/definition/.

[13] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992. URL `citeseer.ist.psu.edu/peytonjones92implementing.html`.

[14] Simon L. Peyton Jones. Compiling haskell by program transformation: A report from the trenches. In *European Symposium on Programming*, pages 18–44, 1996. URL `http://citeseer.ist.psu.edu/peytonjones96compiling.html`.

[15] Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36, 1999. URL `http://citeseer.ist.psu.edu/article/peytonjones98semantics.html`.

[16] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell, 2002. URL `http://research.microsoft.com/Users/simonpj/papers/marktoberdorf/`.

[17] Simon Peyton Jones, Thomas Nordin, and Dino Oliva. C--: A portable assembly language. *Lecture Notes in Computer Science*, 1467, 1998. URL `http://citeseer.ist.psu.edu/341046.html`.

[18] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, 1999. URL `http://citeseer.ist.psu.edu/239439.html`.

[19] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. *ACM SIGPLAN Notices*, 35(5): 285–298, 2000. URL `http://citeseer.ist.psu.edu/ramsey00single.html`.

Appendix A

Exception example in C--

Below the C-- code of the GRIN example shown in Section 4.3 on page 39.

```
target memsize 8 byteorder little pointersize 32 wordsize 32;
typedef bits32 grWord;
grWord eh;
import "GC_malloc" as alloc;
export grin_main     as "grin_main";
const grWord Fmain          = 5;
const grWord Fthr           = 4;
const grWord COutOfBounds = 3;
const grWord CDivByZero     = 2;
const grWord CInt           = 1;
const grWord Frethrow       = 0;
section "data"{ GlobalNodes :
                  main_caf : grWord[5]{ Fmain, 0, 0, 0, 0};
                  GlobalNodesEnd :
                }
grin_main (){
  grWord _, result;
  _, result = main ();
  return (result);
}
main (){
  grWord _;
  grWord r.57, r.58;
  stackdata{ prev.eh.e : grWord; }
  grWord[prev.eh.e], eh = eh, handler.e;          // push catch (e)
  grWord e1;
  e1 = foreign "C" alloc (20);
  grWord[e1] = CDivByZero;
  grWord e2;
  e2 = foreign "C" alloc (20);
  grWord[e2] = COutOfBounds;
  grWord f1;
```

```
    f1 = foreign "C" alloc(20);
    grWord[f1], grWord[%add(f1, 4)] = Fthr, e1;
    grWord f2;
    f2 = foreign "C" alloc(20);
    grWord[f2], grWord[%add(f2, 4)] = Fthr, e2;
    r.57, r.58 = add(f1, f2) also cuts to handler.e;
    eh = grWord[prev.eh.e];                          // pop catch(e)
    goto after.e;
    grWord e;
    continuation handler.e(e):
    eh = grWord[prev.eh.e],;
    r.57, r.58 = CInt, 1;
    after.e:
    return(r.57, r.58);
}
add(grWord a, grWord b){
    grWord _;
    grWord a.26.38;
    a.26.38 = grWord[a];
    switch a.26.38{
      case Frethrow: {
        grWord a.26.40.50;
        a.26.40.50 = grWord[%add(a, 8)];
        cut to eh(a.26.40.50);
      }
      case Fthr: {
        grWord a.26.40.52;
        a.26.40.52 = grWord[%add(a, 8)];
        grWord x54, x55;
        stackdata{ prev.eh.a.29: grWord; }
        grWord[prev.eh.a.29], eh = eh, handler.a.29; // push catch(a.29)
        thr(a.26.40.52) also cuts to handler.a.29;
        eh = grWord[prev.eh.a.29];                   // pop catch(a.29)
        goto after.a.29;
        grWord a.29;
        continuation handler.a.29(a.29):
        eh = grWord[prev.eh.a.29];
        grWord[a], grWord[%add(a, 4)] = Frethrow, a.29;
        cut to eh(a.29);
        after.a.29:
        return(x54, x55);
      }
    }
}
thr(grWord x){
    grWord _;
    cut to eh(x);
}
```