Compiling by transformation: efficient implementation of overloading in Haskell

Jeroen Fokker S. Doaitse Swierstra
Utrecht University
{jeroen,doaitse}@cs.uu.nl

Abstract

The Utrecht Haskell Compiler (UHC) is designed as the composition of many small transformations. We illustrate the transformational approach by showing how overloading is implemented and optimized in UHC. Overloaded functions take additional 'dictionary' arguments, which are automatically inserted during code generation, based on the inferred types.

For each instance declaration, a dictionary is generated containing the functions defined in that instance. The dictionary also contains the default definitions from the corresponding class declaration, thus requiring a mechanism for combining them.

When modules are compiled separately, this combining is done dynamically, during program startup or at the first use of the dictionary. When performing whole-program analysis, however, information from the class and instance declarations can be combined statically using symbolic computation. Further transformations, notably specialization of functions for constant arguments, can completely eliminate the run-time overhead normally associated with dictionary passing.

1. Introduction

Compiling a program is the oldest and most widely used example of generative programming. They are so common, that programmers are hardly conscious any more that compiling, say, a C pogram alleviates them of the tedious task of writing machine code. With compilers for higher level languages, it is more manifest that a compiler generates code, which in turn can be regarded as source code for another compiler. For example, Haskell compilers often target the C language, and domain specific languages in turn may target Haskell.

Compilers are complex programs. The translating process consists of many subtasks, such as parsing, type checking, modelling memory structure, code generation, and optimization. These tasks ideally are dealt with in separate components, and by selecting and arranging these components one can tune a compiler for different requirements, such as various levels of optimization, or targeting various back ends.

[Copyright notice will appear here once 'preprint' option is removed.]

We decided to build a compiler for Haskell (the Utrecht Haskell Compiler, or UHC) that radically takes a component based approach: it translates a program by transforming it in small steps [Dijkstra 2005, Dijkstra, Fokker and Swierstra 2009, website UHC]. Each of the many transformations is relatively simple, and can be understood and tested separately. Along the route from source (Haskell) to target (C) language, the program is transformed through quite a few intermediate languages: some of them existing, some of them especially defined for the purpose. Therefore some transformations (5 of them) are actually translations between languages, and many others (about 60) are source-to-source translations, staying within one of the intermediate languages.

In this paper we present a case study that illustrates the approach: the implementation of function overloading in Haskell. This case touches many aspects: modelling a high-level concept (overloading) by lower level datastructures (dictionaries), dealing with two compiler modes (separate compilation versus whole-program optimization), and the composition of 6 transformations that together accomplish the task.

Moreover, the Haskell feature under scrutiny (type classes and their instances) happens to be a mechanism that is often used by Haskell programmers for achieving modularity, and thus is interesting in its own right from a component engineering perspective.

To make the paper self-contained, we start in section 2 with a short description of overloading in Haskell. In section 3 we sketch the structure of UHC, and introduce the intermediate language central to the case study in this paper. Next we describe the implementations of overloading for two different compiler modes: in separate compilation mode (section 4) we see generative programming at

work, wheras in whole-program analysis mode (section 5) many components cooperate to optimize the code. We conclude with remarks on the methodology and pointers to related work.

2. Overloading in Haskell

In this section we briefly summarize how function overloading is modeled in Haskell [Peyton Jones 2003]. Overloaded functions can be applied to values of various types. For example, the addition function $(+)$ can be applied to both Int values and Float values (but not to Bool values). An equality test function $(==)$ is available for many basic types, such as Int, Float, Char, Bool. Furthermore, it can even be applied to values of list type $[\,a\,]$, provided that it is also defined for the element type a. Note however that equality testing is not possible for values of a function type.

A closely related concept is *polymorphism*. Polymorphic functions can be applied to values of various types. For example, concatenation can be applied to lists of type $[\,\mathit{Int}\,]$ and to lists of type $[\,\mathit{Char}\,]$, and in fact, to lists of type $[\,a\,]$ for any type a.

The difference is that polymorphic functions have a uniform definition, regardless of the type of the arguments, whereas an overloaded function usually has a different definition for each type of argument for which it is defined. The type of arguments of polymorphic functions can be *any* instantiation of the type variables in its type, whereas the type of arguments of overloaded functions can only be those types for which a version of the function has been defined. Therefore, function overloading is sometimes referred to as

ad hoc polymorphism.

2.1 Class and instance declarations

The group of types to which an overloaded function can be applied is known as its *class*. A *class declaration* introduces a name for such a group of types, together the signatures of functions. These signatures can involve type variables, which may be instantiated to types belonging to the group. For example, we can define a class Eq to group the types for which equality and non-equaltiy are defined:

$$
\begin{aligned}
&\textbf{class}\ Eq\ a\ \textbf{where} \\
&\quad eq :: a \rightarrow a \rightarrow Bool \\
&\quad ne :: a \rightarrow a \rightarrow Bool
\end{aligned}
$$

This Eq class is actually part of the Haskell standard library. There, operators == and /= are defined rather than functions eq and ne, but in this paper we avoid operators, as the notational conventions for writing them may obscure the explanation.

Individual types can be made an instance of a class by an *instance declaration*. For example, Int can be made instance of Eq by providing definitions for the functions specified in the class declaration:

$$
\begin{aligned}
&\textbf{instance}\ Eq\ Int\ \textbf{where} \\
&\quad eq\ x\ y = eqPrimInt\ x\ y \\
&\quad ne\ x\ y = not\ (eqPrimInt\ x\ y)
\end{aligned}
$$

The definitions rely on the existence of a function $eqPrimInt$ that gives a primitive implementation for equality on integers. The way $eqPrimInt$ is defined is not relevant for the present discussion.

As another example, we give an instance declaration for $Eq\ Bool$. It shows that we can define the function eq from scratch if we wish,

without relying on other functions. The function ne can be defined in a similar way (and that would probably be more efficient), but we show here the possibility to have ne rely on the existence of the other overloaded function eq.

$$\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ \mathit{Bool}\ \textbf{where} \\
\quad \begin{array}{llll}
\mathit{eq} & \mathit{False} & \mathit{False} & = \mathit{True} \\
\mathit{eq} & \mathit{True} & \mathit{True} & = \mathit{True} \\
\mathit{eq} & _ & _ & = \mathit{False} \\
\mathit{ne}\ x & & y & = \mathit{not}\ (\mathit{eq}\ x\ y)
\end{array}
\end{array}$$

The terminology of the notions ‘class’ and ‘instance’ is borrowed from the object-oriented paradigm [Bernardy et al 2009]. There are similarities, in that a class declaration specifies functions that are implemented by instance declaration. But note that there are differences as well: in the object-oriented paradigm, class member functions take an object of the class as an implicit argument, whereas in the functional paradigm all parameters are declared explicitly; thus we can have two arguments that are constrained to the type of this instance declaration.

Not all overloaded functions need to be defined in a class. Every function that uses an overloaded function, becomes automatically overloaded itself. For example, the function $elem$ that checks element membership of a list uses the overloaded equality function eq on the list elements:

$$\begin{array}{ll}
\mathit{elem}\ e\ [\,] & = \mathit{False} \\
\mathit{elem}\ e\ (x : xs) & = \mathit{eq}\ e\ x \vee \mathit{elem}\ e\ xs
\end{array}$$

Therefore, although the $elem$ function has a uniform definition and thus seems to be a polymorphic function, it can actually only be

used on lists of which the elements are in the *Eq* class. This fact is expressed in the signature of the *elem* function by:

$$elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$$

Note the double-shafted arrow which is to be read as 'constrains', as opposed to the single-shafted arrow which is to be read as 'function from/to'. We may read this signature as: for each type a which is an instance of *Eq*, *elem* has the type $a \rightarrow [a] \rightarrow Bool$.

2.2 Default definitions

One could question the need for declaring *ne* to be part of the *Eq* class. After all, non-equality is (in any sound implementation) the negation of equality. So it would have been possible to only specify *eq* in the class, and to define *ne* with a uniform definition, as we did with *elem*:

```
class Eq a where
  eq :: a -> a -> Bool
  -- and outside the class:
ne ::  Eq a => a -> a -> Bool
ne x y = not (eq x y)
```

Using this approach, instance programmers are freed from the burden to give an explicit, but straightforward, definition of *ne*. But the downside is that it is now impossible for instance programmers to define their own, probably more efficient, version of *ne*. The programmer of *Eq Bool* might regret not being able to define a version of *ne* that uses pattern matching directly, instead of relying on the uniform definition based on *eq* and *not*.

To overcome this dilemma, Haskell allows a class definition to be augmented with *default definitions* for some or all of the member functions. Now, instance programmers have the choice either to

rely on a default definition of ne as it appears in the class, or to give a more efficient version for a particular type.

A default definition by nature is a uniform definition, as it cannot assume the type variables in the signature to be instantiated to a particular type. But the definition can refer to the other functions from the class. In fact, the Haskell standard library implementation of Eq has default definitions for both ne and eq, defined in terms of each other:

$$
\begin{array}{l}
\textbf{class } Eq\ a \textbf{ where} \\
\quad eq :: a \rightarrow a \rightarrow Bool \\
\quad ne :: a \rightarrow a \rightarrow Bool \\
\quad ne\ x\ y = not\ (eq\ x\ y) \\
\quad eq\ x\ y\ = not\ (ne\ x\ y)
\end{array}
$$

An instance programmer now has the choice to define either eq or ne (and rely on the default definition for the other), or define both (thus ignoring/erasing/overriding both default definitions). Defining neither of the two is allowed as well, but would result in inheriting the circular definitions without breaking the circle by redefining at least one of the two.

2.3 Superclasses

Using the terminology of object-oriented paradigm even further, Haskell has the notion of a 'superclass' as well. It is exemplified by the class Ord of types that have an ordering, by providing comparison operators like $<$, $\leqslant$, $>$, and $\geqslant$. Default definitions specify these in terms of each other. Instance programmers can choose to implement only one of the four (and rely on the default definitions for the others), or more if they wish. But the default definitions not only call each other, but also the eq function from

class *Eq*. This is possible because *Ord* is specified to be a subclass

of *Eq*; that is, a type is only allowed to be an instance of *Ord* if it is an instance of *Eq* as well.

A possible class definition of *Ord* is:

$$
\begin{array}{l}
\textbf{class } Eq\ a \Rightarrow Ord\ a \textbf{ where} \\
\quad lt\ :: a \rightarrow a \rightarrow Bool \\
\quad le\ :: a \rightarrow a \rightarrow Bool \\
\quad gt\ :: a \rightarrow a \rightarrow Bool \\
\quad ge\ :: a \rightarrow a \rightarrow Bool \\
\quad lt\ \ x\ y = not\ (ge\ x\ y) \\
\quad gt\ x\ y = not\ (le\ x\ y) \\
\quad le\ \ x\ y = lt\ \ x\ y \vee eq\ x\ y \\
\quad ge\ x\ y = gt\ x\ y \vee eq\ x\ y
\end{array}
$$

The superclass is mentioned in the class header, featuring another meaning of the double-shafted arrow. There can be more than one superclass, separated by commas and enclosed in parentheses. The actual definition in the standard library also specifies functions *compare*, *min*, and *max*, and an even more intricate web of mutually recursive default definitions.

2.4 Context for instances

Finally, Haskell provides a mechanism to declare an instance in the context of another instance. An example is the definition of *Eq* for lists, where the equality for lists of elements is expressed using the equality function for the elements. This only makes sense in a

context where the element type is assumed to be instance of Eq as well.

$$\begin{array}{l}\textbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where}\\ \quad eq\ []\ \ []\ = True\\ \quad eq\ []\ \ (y : ys) = False\\ \quad eq\ (x : xs)\ (y : ys) = eq\ x\ y \wedge eq\ xs\ ys\end{array}$$

The defining expression line contains two calls to eq. The second is a recursive call on the tails of the list. The first however is not a recursive call, but a call to eq for a different instance type: the list element type, guaranteed to be an instance of Eq by the context mentioned in the header. Type analysis by the compiler makes sure that the right version of eq is called.

3. UHC compiler structure

3.1 Transformational programming

The main structure of the Utrecht Haskell Compiler is shown in Figure 1. Haskell source text is translated to an executable program by stepwise transformation. Some transformations translate the program to a lower level language, many others are transformations within one language, establishing an invariant or performing an optimization. A more detailed account is given in a separate paper [Dijkstra, Fokker and Swierstra 2009].

All transformations, both within a language and between languages, are expressed as an algebra giving a semantics to the language. The algebras are described with the aid of an attribute grammar, which makes it possible to write multi-pass tree-traversals without even knowing the exact number of passes. Although the

compiler driver is set up to pass data structures between transformations, for all intermediate languages we have a concrete syntax with a parser and a pretty printer. This facilitates debugging the compiler, by inspecting code between transformations. Here is a short characterization of the intermediate languages:

- Haskell (HS): a general-purpose, higher-order, polymorphically typed, lazy functional language.

module1 *module2*

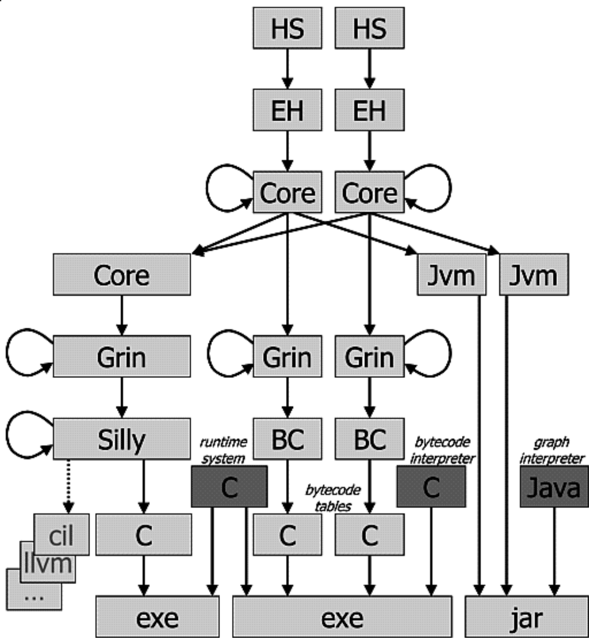

Figure 1. Intermediate languages and transformations in the UHC pipeline, in each of the three operation modes: whole-program analysis (left), bytecode interpreter (middle), and Java (right).

- Essential Haskell (EH): a higher-order, polymorphically typed,

lazy functional language close to lambda-calculus, without syntactic sugar.

- Core: an untyped, lazy functional language close to lambda-calculus, augmented with let-bindings and case distinction with simple pattern matching.
- Grin: 'Graph reduction intermediate notation', the instruction set of a virtual machine of a small functional language with strict semantics, with features that enable implementation of laziness [Boquist 1999].
- Silly: 'Simple imperative little language', an abstraction of features found in every imperative language (if-statements, assignments, explicit memory allocation) augmented with primitives for manipulating a stack, easily translatable to e.g. C (not all features of C are provided, only those that are needed for our purpose).
- BC: A bytecode language for a low-level machine intended to interpret Grin which is not whole-program analyzed nor transformed. We do not discuss this language in this paper.

The compiler targets different back ends, based on a choice of the user. In all cases, the compiler starts compilation on a per module basis, desugaring the Haskell source text to Essential Haskell, type checking it and translating it to Core. Then there is a choice from three modes of operation:

- In *whole-program analysis mode*, the Core modules of the program and required libraries are assembled together and processed further as a whole. At the Grin level, elaborate intermodule optimization takes place. Ultimately, all functions are translated to low level C, which can be compiled by a standard compiler. As alternative back ends, we are experimenting with other target languages, among which are the Common Intermediate Language (CIL) from the Common language infrastruc-

ture used by .NET, and the Low-Level Virtual Machine (LLVM) compiler infrastructure.

$$
\begin{array}{lcl}
\mathit{Program} & ::= & \mathit{Bind}^{*} \\
\mathit{Bind} & ::= & \underline{\mathit{Name}}\ \underline{\mathit{Var}}^{*} = \mathit{Annot}\ \{\mathit{Expr}\} \\
 & \mid & \underline{\mathit{Var}} \leftarrow \textbf{store}\ \mathit{Node} \\
\mathit{Expr} & ::= & \textbf{unit}\ \mathit{Var} \\
 & \mid & \textbf{unit}\ \mathit{Node} \\
 & \mid & \mathit{Expr};\lambda \underline{\mathit{Var}} \rightarrow \mathit{Expr} \\
 & \mid & \textbf{store}\ \mathit{Node} \\
 & \mid & \textbf{call}\ \mathit{Name}\ \mathit{Var}^{*} \\
 & \mid & \textbf{apply}\ \mathit{Var}\ \mathit{Var}^{*} \\
 & \mid & \textbf{eval}\ \mathit{Var} \\
 & \mid & \textbf{case}\ \mathit{Var}\ \textbf{of}\ \mathit{Alt}^{*} \\
\mathit{Alt} & ::= & \mathit{Pat} \rightarrow \mathit{Expr} \\
\mathit{Pat} & ::= & (\mathit{Tag}\ \underline{\mathit{Var}}^{*}) \\
\mathit{Node} & ::= & (\mathit{Tag}\ \mathit{Var}^{*}) \\
 & \mid & (\mathit{Tag}\ \mathit{Lit}) \\
\mathit{Tag} & ::= & \mathbf{C}/\mathit{Constr} \\
 & \mid & \mathbf{P}_{N}/\mathit{Name} \\
 & \mid & \mathbf{F}/\mathit{Name} \\
 & \mid & \mathbf{A} \\
N & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
\mathit{Annot} & ::= & \epsilon \\
 & \mid & \textbf{dictclass}\ (\mathit{Name}^{*}) \\
 & \mid & \textbf{dictinst}\ (\mathit{Constr}\ \mathit{Name}\ (\mathit{Name}^{*}))
\end{array}
$$

| **overloaded** $((N^*)^*)$
| **specialized** $(Name\ (Var^*))$

Figure 2. Syntax of the Grin language. An * denotes 0 or more occurrences. *Var*, *Name*, and *Constr* are identifiers referring to values, functions, or constructors. Underlining denotes a defining position. *Lit* is an integer or character literal.

- In *bytecode interpreter mode*, the Core modules are translated to Grin separately. Each Grin module is translated into instructions for a custom bytecode machine. The bytecode is emitted in the form of C arrays, which are interpreted by a handwritten bytecode interpreter in C.
- In *Java mode*, the Core modules are translated to Java bytecode, to be interpreted by the Java virtual machine (JVM). Each function is translated to a separate class with an *eval* function, and each closure is represented by an object combining a function with its parameters. Together with a driver function in Java which steers the interpretation, these can be stored in a Java archive (jar) and be interpreted by a standard Java interpreter.

The bytecode interpreter mode is intended for use during program development: it takes less time to compile, but because of the interpretation overhead the resulting program runs slower. The whole-program analysis mode is intended to be used for the final program: it takes more time to compile, but generates code that is more efficient.

3.2 The Grin intermediate language

Grin is a small language designed to be an intermediate language between Core (a higher-order lazy functional language) and Silly (an imperative language). Grin is a functional language (function bodies are expressions which are evaluated when a function is called) but it is first-order (all functions are defined at top level and cannot be passed as arguments) and strict (arguments are evaluated when functions are called). Nevertheless, Grin has an imperative spirit, as the evaluation of expressions can have side effects on the heap. The sequencing of these side effects is made explcit in the

Grin code, using a monadic notation: the only possible expression forms are a handful of basic operations on the heap, and a monadic bind and unit operation.

The mapping from Core to Grin makes the evaluation order explicit. Thus, it has to simulate the lazy semantics of Core by means of primitives that are provided in Grin for that purpose. The data that is manipulated by Grin is structured into *nodes*, which are tagged lists of single-word elements. Nodes can be assigned to local variables directly, but local variables can also hold pointers to nodes that are stored on the heap.

Refer to figure 2 for the syntax of the Grin language. It shows that a program consists of bindings, each of which is either a function definition or the initialization of global variable with a pointer to a node stored on the heap. Function definitions are optionally annotated with an $Annot$, the meaning of which is described in section 5.

The body of a function is an expression of the built-in monad: ei-

ther one of six primitive forms, a monadic binding of one expression to a variable in another expression, or a selection from alternatives based on the value of a variable. Sequencing is written in monadic style as $e_1; \lambda x \rightarrow e_2$, with semantics 'execute e_1, bind the result to x and then execute e_2 in the new context. Although the lambda symbol suggests that we can define local functions, this is by no means general lambda-calculus, as the lambda can only be used in this particular position. The notation involving the semicolon, lambda and arrow is just a triadic expression form, which in a different concrete syntax would look more like an imperative assignment statement $x := e_1; e_2$.

We will discuss the semantics of the various expression forms shortly, but first we focus on nodes and tags. A node can be thought of as a variable-sized block of memory, consisting of a tag indicating its purpose, and a payload of zero or more pointers. A special form of node consists of a tag and a primitive literal.

Tags come in four flavors: **C**, **P**, **F**, and **A**. Nodes with **C** or **P** tag are final: when a pointer to them is evaluated, they remain the same. Nodes with **F** or **A** tags encode a deferred computation. They are also known as *thunks*, used in the implementation of lazy evaluation. When a pointer to such a node is evaluated, the computation takes place, and the node is updated with a final node.

The meaning of the four tag flavors is as follows:

- **C** corresponds to a constructor in Haskell. For example, the empty list has tag **C**/*Nil* and can be represented as a node with zero-length payload (**C**/*Nil*). A non-empty list can be constructed as (**C**/*Cons* x xs).
- **P** stands for a partially parametrised function. In the lambda calculus, a function application with to few arguments to be reduceable is in *head normal form*, that is, cannot be further

evaluated. The **P** tag encodes which function was partially parameterized, and (written as a numeric suffix) the number of parameters it still lacks. The payload of a node with **P**-tag holds the argunents it has already received. For example, the node $(\mathbf{P}_1/\mathit{plus}\ \mathit{one})$, where *one* could be globally bound by $\mathit{one} \leftarrow$ **store** $(\mathbf{C}/\mathit{Int}\ 1)$, denotes a partial parameterization of function *plus*, still lacking its second argument.

- **F** is used to denote a deferred call to a function. For example, the node $(\mathbf{F}/\mathit{divide}\ \mathit{one}\ \mathit{zero})$ is an encoding of the fact that function *divide* will be called, but only when this node is forced to evaluation.
- **A** is used to denote the deferred application of a partially parameterized function to further arguments. Nodes with this tag must have a payload of at least one pointer, which is supposed to evaluate to a node with **P** tag. For example, the node $(\mathbf{A}\ \mathit{succ}\ \mathit{one})$, where variable *succ* is bound to node

$(\mathbf{P}_1/\mathit{plus}\ \mathit{one})$, will be updated to $(\mathbf{C}/\mathit{Int}\ 2)$, but only when forced to evaluation.

We now turn to the semantics of the various expression forms.

- **unit** n, where n is an explicit node, just returns that node, without having side effects on the heap.
- **unit** x just returns the value of the variable x, which can be a node or a pointer value.
- $e_1; \lambda x \rightarrow e_2$ executes e_1, binds the result to variable x and subsequently executes e_2. It returns whatever e_2 returns.
- **store** n stores the node value n on the heap, and returns a pointer to it

- **call** $f\ a_1 \ldots a_n$ calls function f with the given arguments, and returns the node that is its result
- **apply** $x\ b_1 \ldots b_k$ expects variable x to evaluate to a node with value $(\mathbf{P}_m/f\ a_1 \ldots a_n)$, and calls $f\ a_1 \ldots a_n\ b_1 \ldots b_m$ if $k \geqslant m$. When $k > m$, the result of the call is recursively applied to the remaining arguments.
  When $k < m$, the function can't be called for lack of arguments, and an appropriate new $\mathbf{P}_{m-k}$-node is returned.
- **eval** x expects a pointer in variable x and fetches the corresponding node from the heap. If the node is in head normal form, that is, has a **C** or **P** tag, that node is just returned. If the node has an **F**/f tag, function f is called. The node is updated with the result, and that result is also returned. If the node is $(\mathbf{A}\ x\ a_1 \ldots a_n)$, the effect is the same as an **apply**.
- **case** x **of**$\ldots p\ \rightarrow\ e\ldots$ expects a node in variable x and executes expression e coupled with pattern p that matches that node.

Variables in a Grin program can in principle hold either a pointer or a node. But all syntactic structures involving variables have specific requirements on the value of that variable. In this sense Grin is a (implicitly) typed language: it can be statically determined which variables hold pointers, and which hold nodes. The type requirements are as follows: the first *Var* in an **apply** and the one in a **case** should hold a node, but the other *Var*s occurring in a **eval**, **call**, **apply**, and node payload should hold a pointer. The *Var* in a **unit** can hold either a pointer or a node.

Likewise, the return value of each statement form is fixed: **call**, **apply**, **eval**, and **unit** n return a node, whereas **store** returns a pointer. The remaining expression forms **unit**, sequencing, and **case**, return whatever their sub-constructs return.

The code generator (Core to Grin transformation) takes care that the generated Grin program is type correct in this respect. It can generate a **store** expression when a node needs to be available as a pointer, and it can generate an **eval** expression when the node pointed to by a pointer is needed. For a final node, **eval** just fetches the node pointed to. For a non-final (thunk) node, **eval** first forces the thunk to a final node, but that is a good thing to do when a node is needed.

4. Dynamic handling of overloading

4.1 Dictionaries

The standard technique for implementing overloading in Haskell, as described by Augustsson [Augustsson 1993], makes use of *dictionaries*. A dictionary basically contains the implementations for a particular instance of the functions specified in a class. Calling an overloaded function then amounts to selecting one of the functions from the dictionary, and subsequently applying it to its arguments.

Function names are not storable values in Grin, but we can store a node that represents a partially parameterized function. As an example, consider the instance declaration in Haskell that defines both member functions of *Eq* for the *Int* type:

instance $Eq\ Int$ **where**
$eq\ x\ y = eqPrimInt\ x\ y$
$ne\ x\ y = not\ (eqPrimInt\ x\ y)$

A Grin representation of the dictionary corresponding to this dec-

laration is a node, using the dictionary name as a constructor, and having two pointers corresponding to the two function definitions as its payload: The two pointers point to nodes denoting partial parameterizations of functions: they have 'already' received 0 arguments, and are still lacking 2 arguments:

$$
\begin{aligned}
&\mathit{eqIntP} &&\leftarrow \textbf{store}\ (\mathbf{P}_2/\mathit{eqPrimInt})\\
&\mathit{neIntP} &&\leftarrow \textbf{store}\ (\mathbf{P}_2/\mathit{neInt})\\
&\mathit{dictEqInt} &&\leftarrow \textbf{store}\ (\mathbf{C}/\mathit{Eq}\ \mathit{eqIntP}\ \mathit{neIntP})
\end{aligned}
$$

As the definition of *eq* for *Int* is just a call to *eqPrimInt*, we can use $(\mathbf{P}_2/\mathit{eqPrimInt})$ for the first pointer. But the definition of *ne* for *Int* is not just a renaming of an existing function. It therefore gives rise to a new function binding in Grin:

$$
\begin{aligned}
&\mathit{neInt}\ x\ y =\\
&\{\textbf{store}\ (\mathbf{F}/\mathit{eqPrimInt}\ x\ y); \lambda r \rightarrow\\
&\ \ \textbf{call}\ \mathit{not}\ r\\
&\}
\end{aligned}
$$

The second member of the *dictEqInt* dictionary points to the $\mathbf{P}_2$-node of this function.

The class declaration in Haskell, which only contains signatures for the member functions, nevertheless gives rise to generated Grin code: one function binding for each member function. These functions take a single dictionary argument, and select the appropriate field from that dictionary. The selector functions for the *Eq* class are polymorhpic functions that pick a specific element from a dictionary:

$$
\begin{aligned}
&\mathit{eq}\ d =\\
&\{\textbf{eval}\ d; \lambda t \rightarrow\\
&\ \ \textbf{case}\ t\ \textbf{of}\\
&\ \ \ \ \ (\mathbf{C}/\mathit{Eq}\ p\ q) \rightarrow \{\textbf{eval}\ p\}
\end{aligned}
$$

```
}
ne d =
{eval d; λt →
  case t of
     (C/Eq p q) → {eval q}
}
```

Finally, we describe the code that is generated for the call in Haskell of a class member function, as in the example

$test1 = eq\ 3\ 4$

Grin code for this example does the call to the member function in two steps. First the selector function is called on the dictionary corresponding to the type (*Int* in the example) of the arguments. This is supposed to return a $\mathbf{P}_2$-tagged node, which is subsequently applied to the arguments by **apply**.

```
test1 =
{store (C/Int 3)  ; λx →
  store (C/Int 4)  ; λy →
  call eq dictEqInt; λp →
  apply p x y
}
```

Overloaded function that are not declared in the class, such as the Haskell function

$elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$

are implemented as a Grin function that takes an additional dictionary argument. In essence, the overloaded function is made polymorphic again.

4.2 Default definitions

Now suppose that we are compiling a class with a default definition, for example the *Eq* class with a default definition for *ne*:

```
class Eq a where
  eq :: a → a → Bool
  ne :: a → a → Bool
  ne x y = not (eq x y)
```

Clearly, the compiler needs to generate code for the default function. It can't be named *ne*, as that name is already used for the selector function, so let's name it *neDef*. In the body of *neDef* we need to call *eq*. This process, as in the *test1* example above, takes two steps: first select the function from the dictionary, then apply it to its arguments.

The dictionary where the member functions can be found needs to be passed as an additional argument to *neDef*. Essentially, default functions are implemented just as overloaded functions outside a class, such as *elem*. So, although the Haskell default definition for *ne* has 2 parameters, its Grin implementation has 3:

```
neDef d x y =
{store (F/eq d)  ; λf →
 store (A f x y); λr →
 call not r
}
```

Note that the 2-stage calling process is disguised here in its lazy form. Instead of a **call** and an **apply** as in the *test1* example, we

store a deferred call to the selector by means of an **F**-node, and a deferred apply by means of an **A**-node. Whether or not these thunk-nodes are ever evaluated is decided inside the *not* function. Actually, *not* indeed does evaluate its argument, but without a strictness analysis we can't predict that when compiling *neDef*.

Inside a dictionary for class *Eq*, we need two $\mathbf{P}_2$-nodes, as *eq* and *ne* in Haskell have 2 arguments. We would get something like:

$$\begin{aligned} &eqIntP && \leftarrow \textbf{store}\ (\mathbf{P}_2/eqPrimInt) \\ &neIntP && \leftarrow \textbf{store}\ (\mathbf{P}_2/neDef\ dictEqInt) \\ &dictEqInt && \leftarrow \textbf{store}\ (\mathbf{C}/Eq\ eqIntP\ neIntP) \end{aligned}$$

The dictionary is to be populated with pointers to **P**-nodes, that refer to functions which are a mixture from the instance declaration (*eqIntP* in the example) and default definitions from the class declaration (*neIntP*). Note that *neIntP* and *dictEqInt* are referring to each other, which is allowed in Grin.

In a setting where compilation is done for separate modules, it is not possible to generate without extra tricks the dictionary at compile time. Class and instance declarations can reside in separate modules, and normally the code generated for the class declaration is not available for inspection by the compiler at the moment that the *dictEqInt* binding is generated.

Instead of having the compiler emit the binding

$$dictEqInt \leftarrow \textbf{store}\ (\mathbf{C}/Eq\ eqIntP\ neIntP)$$

directly, the compiler generates code that can construct the dictionary at run time, at the first occasion that it is used.

4.3 Dynamic generation of dictionaries – first attempt

Instead of the explicit construction of the dictionary by the com-

piler, the compiler generates a nullary function that can construct it on demand. The dictionary variable is bound to a thunk for that function, thus triggering the construction when the dictionary is first used.

$makeEqInt =$
{ -- construct P2-nodes for both dictionary fields

}
$dictEqInt \leftarrow$ **store** ($\mathbf{F}/makeEqInt$)

By the nature of the evaluation/updating mechanism, from the moment that *dictEqInt* has been evaluated for the first time, it will be bound to the final $\mathbf{C}/Eq$-node.

For construction the dictionary, we need information from the class declaration. The only way of communication with another mdule is to call functions or to evaluate global bindings. We arrange that the class module, for this purpose, defines a global binding to a dictionary that contains fields for the default functions, and $\perp$ elsewhere.

$neDefP \leftarrow$ **store** ($\mathbf{P}_3/neDef$)
$dictEqDef \leftarrow$ **store** ($\mathbf{C}/Eq \perp neDefP$)

Note that this 'default dictionary' stores $\mathbf{P}_3$-nodes, not $\mathbf{P}_2$-nodes as we need in the instance dictionaries.

Now the dictionary for the instance is constructed by fetching the default dictionary, and applying the $\mathbf{P}_3$-node found there to the dictionary it needs. That is: the dictionary that we are about to build; lazy evaluation allows this seemingly circular definition.

Applying a $\mathbf{P}_3$-node to one further argument results in a $\mathbf{P}_2$-node, which is stored in the dictionary, along with a $\mathbf{P}_2$-node for the other function:

$$
\begin{array}{l}
\mathit{makeEqInt} = \\
\{\mathbf{eval}\ \mathit{dictEqDef};\lambda d' \\
\quad \mathbf{case}\ d'\ \mathbf{of} \\
\qquad (\mathsf{C}/\mathit{Eq}\ _\ q) \rightarrow \\
\qquad \{\mathbf{eval}\ q \qquad\qquad\qquad ;\lambda q' \\
\qquad\ \ \mathbf{apply}\ q'\ \mathit{dictEqInt} \quad ;\lambda q'' \\
\qquad\ \ \mathbf{store}\ (\mathbf{P}_2/\mathit{eqPrimInt});\lambda p \\
\qquad\ \ \mathbf{unit}\ (\mathsf{C}/\mathit{Eq}\ p\ q'') \\
\qquad \} \\
\}
\end{array}
$$

4.4 Dynamic generation of dictionaries

Although the idea in the previous subsection works in simple situations, such as the *Eq* example, it does not work in general. The flaw is that default definitions not always take the additional dictionary parameter as suggested in the first attempt. Situations where that assumption doesn't hold are:

- the default definition does not use the other members. The Core code generator only introduces additional parameters when they are really necessary, so here we get a default definition without dictionary parameter.
- the default definition does use other fields, but only from superclasses. The Core code generator optimizes for this situation by having the function take the dictionary for the superclass di-

rectly.

In both situations the expression **apply** q' $dictEqInt$ in the first attempt applies q' wrongly.

To solve this problem, we take an approach inspired by Faxén [Faxén 2002]. His endeavour was to give a formal definition of the semantics of Haskell, but actually the construction used there works quite well in a practical setting.

The idea is that the responsibility for applying the default dictionary to the (thunk for the) final result is shifted from the instance module to the class module. In this way the class module, that knows about the expectations of the default definitions, can decide to apply the default definition to the final result, not to do that if it is not needed, or rather use the superclass when needed.

2010/5/31

For this purpose, the creation of the default dictionary is now made dynamic as well, and parameterized by the instance that it is needed in:

$$
\begin{array}{l}
makeEqDef\ d = \\
\{\textbf{store}\ (\mathbf{P}_2/neDef\ d); \lambda q \rightarrow \\
\ \ \textbf{unit}\ (\mathbf{C}/Eq \perp q) \\
\}
\end{array}
$$

In this example, in the first line it was decided to apply the definition to the final dictionary after all, but here the compiler has freedom to act differently if the desires of $neDef$ would have been different.

In this new arrangement, we revise the definition of $makeEqInt$ such that, instead of fetching $eqDef$, it calls our new dynamic

generator *makeEqDef*:

$$
\begin{array}{l}
\quad makeEqInt = \\
\quad \{\textbf{call}\ makeEqDef\ \ dictEqInt; \lambda d' \\
\quad\ \ \textbf{case}\ d'\ \textbf{of} \\
\quad\quad\quad (\mathbf{C}/Eq\ _\ q) \rightarrow \\
\quad\quad\quad \{\textbf{store}\ (\mathbf{P}_2/eqPrimInt); \lambda p \\
\quad\quad\quad\ \ \textbf{unit}\ (\mathbf{C}/Eq\ p\ q) \\
\quad\quad\quad \} \\
\quad \}
\end{array}
$$

5. Static handling of overloading

When it is possible to inspect and transform the program as a whole, we can fully eliminate the run-time overhead of manipulating dictionaries. The idea was first described by Jones [Jones 1995]. We take a radical transformational approach, describing the necessary steps as separate program transformation steps. Thus we adhere to our philosophy of preferring a large number of easy transformations, over a small number of complicated ones.

The most important transformations are:

- MergeInstance: for each instance declarations, merge the definitions found there with the default definitions in the class declaration.
- SelectMember: statically rather than dynamically select members from a dictionary
- SpecConst: specialize functions that are called with a constant argument. This is a general technique that can also transform an expression like $plus\ x\ 1$ to $succ\ x$, where $succ$ is a specialized version of $plus$ with constant argument 1. Here we use it to

specialize overloaded functions that are called with a constant dictionary.

The opportunities for applying these transformations are prepared by some more transformations:

- EvalKnown: simplify uses of **eval** x in a situation where the value of x happens to be statically known
- ApplyKnown: simplify uses of **apply** p x in a situation where the value of p happens to be statically known
- DropUnused: remove bindings to (local and global) variables that are never used
- DropUnreachable: remove bindings to global variables and functions that are not reachable from $main$

5.1 The MergeInstance transformation

In a whole-program analysis setting, it is possible to statically merge the functions defined in an instance with the default definitions from the class. However, the (EH to Core) code generator generates Core code without having a particular back end in mind, and thus we are confronted with the code as described in the previ-

ous section for dynamic dictionary construction. Our first task is to turn that in a static construction. The goal is to replace

$$dictEqInt \leftarrow \mathbf{store}\ (\mathbf{F}/makeEqInt)$$

back to

$$dictEqInt \leftarrow \mathbf{store}\ (\mathbf{C}/Eq\ eqIntP\ neIntP)$$

and to generate the appropriate member fields:

$$eqIntP \leftarrow \mathbf{store}\ (\mathbf{P}_2/eqPrimInt)$$

$neIntP \leftarrow \mathbf{store}\ (\mathbf{P}_2/neDef\ \ dictEqInt)$

This is hard for two reasons: it is hard to see that *dictEqInt* is indeed a dictionary: it would involve deconstructing the thunk, inspecting the code of *makeEqInt* recognize that it has the form of generated code for instance declarations, extract the names of function definitions from it, as well as the reference to *makeEqDef* which is to be deconstructed as well.

Although this approach is possible in principle, it feels like reversely engineering the outcome of all the transformations that were responsible of generating this Grin definitions. Apart from being tricky, the procedure is deemed to break whenever we would make changes in the Core to Grin transformation pipeline.

Instead, we take a different approach, by making the intention of some of the generated function definitions manifest, by means of annotations. The price is that we need to extend both the Core and the Grin language to facilitate such annotations.

- the definition of *makeEqInt* is annotated with a marker **dictinst**: 'this constructs a dictionary corresponding to a instance declaration';
- the definition of *makeEqDef* is annotated with a marker **dictclass**: 'this is a dictionary generator corresponding to a the default definitions in a class declaration';
- the definition of *neDef* is annotated with a marker **overloaded**: 'this is an overloaded default definition with specific additional argument desiderata'.

Apart from the marker we embed in the annotation all information relevant for the dictionaries:

- for **dictinst**, we need the name of the tag of the dictionary, the name of the dictionary constructor for the default definitions, and all the names of the members defined.
- for **dictclass**, we need the names of all default definitions.

- for **overloaded**, we need a description of the dictionaries it needs (and whether it needs one at all).

In our example, the relevant annotations are:

$makeEqInt =$
  dictinst $(Eq\ makeEqDef\ (eqPrimInt\ _))\ \{\dots\}$
$makeEqDef\ d =$
  dictclass $(_\ neDef)\ \{\dots\}$
$neDef\ d\ x\ y =$
  overloaded $(())\ \{\dots\}$

There are underscores in the **dictinst** and **dictclass** annotations for members that are not defined. The annotation of the overloaded function is a list of lists of numbers. Each element corresponds to a dictionary argument. An empty element (as in the example) indicates that this function simply needs the dictionary for the whole class. A singleton list element (n) indicates that a dictionary for the nth superclass is needed. Longer lists specify super-superclasses; for example $(3\ 1)$ specifies the third superclass of the first superclass.

The annotations can be easily inserted when generating Core, because all this information is available anyway when generating the Core definitions. The annotations are propagated unchanged through all Core and Grin transformations, so that we have them

available when we need them: in the MergeInstance transformation.

5.2 The SelectMember transformation

Dictionaries are passed as additional arguments to overloaded functions. In the function bodies, they can be passed to other overloaded functions, but in the end dictionaries are only used for a single purpose: to select a member function from them. An example is *test1* which we introduced earlier:

$$
\begin{array}{l}
\mathit{test1} = \\
\{\textbf{store}\ (\textbf{C}/\mathit{Int}\ 3)\ \ ;\lambda x \rightarrow \\
\ \ \textbf{store}\ (\textbf{C}/\mathit{Int}\ 4)\ \ ;\lambda y \rightarrow \\
\ \ \textbf{call}\ \mathit{eq}\ \mathit{dictEqInt};\lambda p \rightarrow \\
\ \ \textbf{apply}\ p\ x\ y \\
\}
\end{array}
$$

The third line selects a field from the dictionary by calling the *eq* selector; the resulting function is subsequently applied to its arguments.

Now that the previous transformation has made all dictionaries statically available, we can proceed by selecting the member fields statically. This is what the SelectMember transformation does: it scans the Grin program for expressions of the form **call** $s\ d$ where s is a selector function and d is a dictionary. In the example, the selector is *eq*, which is a Grin function defined by

$$
\begin{array}{l}
\mathit{eq}\ d = \\
\{\textbf{eval}\ d; \lambda t \rightarrow \\
\ \ \textbf{case}\ t\ \textbf{of} \\
\ \ \ \ \ \ (\textbf{C}/\mathit{Eq}\ p\ q) \rightarrow \{\textbf{eval}\ p\} \\
\}
\end{array}
$$

It is recognized as a selector, because its definition is a two-line function where the second line evaluates one of the members of a dictionary. We actually hunt the program for selectors, that is functions that have this very structure. (Again this is a form of reverse

engineering; another approach would be to explicitly annotate selector functions as such at the moment they are generated, in a similar fashion as we use **dictinst** and **dictclass** annotations to avoid hunting for complex patterns).

In the example, the dictionary is *dictEqInt*, which is a global variable that at this time (after the MergeInstance transformation) is defined as

$$dictEqInt \leftarrow \textbf{store}\ (\mathbf{C}/Eq\ eqEqInt\ neEqInt)$$

Now that the selector and the dictionary are identified, the call can be performed statically: the expression **call** *eq dictEqInt* is replaced by **eval** *eqEqInt*.

Our example ends up as transformed to:

$$
\begin{array}{l}
test1 = \\
\{\textbf{store}\ (\mathbf{C}/Int\ 3); \lambda x \rightarrow \\
\ \ \textbf{store}\ (\mathbf{C}/Int\ 4); \lambda y \rightarrow \\
\ \ \textbf{eval}\ eqEqInt\quad ; \lambda p \rightarrow \\
\ \ \textbf{apply}\ p\ x\ y \\
\}
\end{array}
$$

5.3 The EvalKnown transformation

At this point in the pipeline of transformations it is useful to perform two transformations that simplify the Grin program based on variables of which the value may be known in a particular context. There are two such transformations: EvalKnown and ApplyKnown.

An **eval** expression occurs in Grin code when it is needed to force a variable to head normal form and to fetch its value from the heap. A

typical occurrence is in the body of a function, where the unknown value of the argument needs to be forced and fetched in order to be scrutinized:

$f\ x =$
$\{\mathbf{eval}\ x; \lambda v \rightarrow$
$\quad \mathbf{case}\ v\ \mathbf{of} \ldots$

Sometimes, a Grin program evaluates a variable of which the value *is* known. We can encounter expressions like:

$\mathbf{store}\ (\mathbf{C}/Int\ 5); \lambda n \rightarrow$
$\mathbf{eval}\ n$

This is equivalent to the shorter expression:

$\mathbf{unit}\ (\mathbf{C}/Int\ 5)$

which is more efficient since it avoids storing a node on the heap, and executing the **eval** operation to fetch it back.

So is the Grin code generator to blame for emitting inefficient code like in the example above? Not really, because the value of the variable could have become known only later, as a result of transformation of Grin code. For example, if the *inline* transformation decides to inline the call to f in

$\mathbf{store}\ (\mathbf{C}/Int\ 5); \lambda n \rightarrow$
$\mathbf{call}\ f\ n$

we end up with such inefficient **store-eval** combinations.

To compensate for this, we introduced a transformation EvalKnown that hunts for occurrences of **eval** x where x has a known value, either because it is a global variable or because it is the target of an earlier **store**. This transformation catches situations like the example above. Since this transformation is carried out anyway, the Grin

code generator can afford itself to generate inefficient **store-eval** pairs on occasion. This makes the code generator simpler: it can be defined compositionally, without having to bother about avoiding **store-eval** pairs.

The EvalKnown transformation symbolically collects all (local and global) **stores**, and for each **eval** x checks whether the variable x has a known value. We distinguish two cases:

- x is a global or local variable used to store a value v in head normal form, that is a node with a **C** or **P** tag. Then **eval** x can be replaced by **unit** v.
- x is a local variable used to store a thunk with an **F** tag, that is a node ($\mathbf{F}/f\ a_1 \ldots a_n$). When x is used only once, then **eval** x can be replaced by **call** $f\ a_1 \ldots a_n$

The reason that we bring up this whole story, is that the previous SelectMember transformation generates opportunities for EvalKnown. Remember that it has replaced **call** *eq dictEqInt* by **eval** *eqEqInt*. This is an opportunity for the EvalKnown transformation, since *eqEqInt* is a global variable bound to ($\mathbf{P}_2/eqPrimInt$). Thus **eval** *eqEqInt* is transformed to **unit** ($\mathbf{P}_2/eqPrimInt$).

Our example thus now is transformed to:

$$
\begin{array}{l}
test1 = \\
\{\mathbf{store}\ (\mathbf{C}/Int\ 3) \quad ; \lambda x \rightarrow \\
\ \mathbf{store}\ (\mathbf{C}/Int\ 4) \quad ; \lambda y \rightarrow \\
\ \mathbf{unit}\ (\mathbf{P}_2/eqPrimInt); \lambda p \rightarrow \\
\ \mathbf{apply}\ p\ x\ y \\
\}
\end{array}
$$

5.4 The ApplyKnown transformation

The **apply** operation expects a value that represents a partially applied function, and applies it to further arguments. Normally this

operation is generated by the code generator for values that are not statically known, for example when emitting code for polymorphic functions such as map.

But similar to the previous subsection, where **eval** is occasionally used on variables with a statically known value, situations can occur where **apply** is used on values that are statically known. In fact, this happens in the example outcome of the previous transformation: we have an **apply** operating on a value that obviously is $(\mathbf{P}_2/\mathit{eqPrimInt})$, a partial application of $\mathit{eqPrimInt}$ lacking 2 parameters.

Since in this example the lacking 2 parameters are provided as part of the **apply** operation, the call is thereby saturated and equivalent to **call** $\mathit{eqPrimInt}\ x\ y$.

This is exactly what the EvalKnown transformation performs: it symbolically collects all **units**, and for each **apply** $x\ b_1 \ldots b_k$ checks whether the variable x holds a known node $(\mathbf{P}_m/f\ a_1 \ldots a_n)$. There are three cases:

- $k < m$ (undersaturated call): replace the **apply** operation by **unit** $(\mathbf{P}_{m-k}/f\ a_1 \ldots a_n\ b_1 \ldots b_k)$
- $k = m$ (saturated call): replace the **apply** operation by **call** $f\ a_1 \ldots a_n\ b_1 \ldots b_k$
- $k > m$ (oversaturated call): do nothing. (This situation does not occur in the context discussed in this paper. If it does occur in other situations, doing nothing is always safe.)

Our example ends up as:

$\mathit{test1} =$
$\{$**store** $(\mathbf{C}/\mathit{Int}\ 3)$ $\quad ; \lambda x \rightarrow$

```
  store (C/Int 4)       ; λy →
  unit (P₂/eqPrimInt); λp →
  call eqPrimInt x y
 }
```

Note that the binding of the $\mathbf{P}_2$-node to p has now become obsolete. However, it is not removed by the ApplyKnown transformation, as it will be caught anyway by a DropUnused transformation performed further downstream the transformation pipeline. This way, we keep the individual transformations straightforward, while they together still do all that is needed.

5.5 The SpecConst transformation

The combined effort of all transformations so far has succeeded in annihilating all dictionary overhead involved in the Haskell expression *eq* 3 4. Now let's see what happens for the Haskell expression *ne* 5 6.

Because the instance declaration *Eq Int* relies on the default definition for *ne*, the resulting Grin code is different. The field selection and subsequent **apply** is shortcut successfully, but the default definition *neDef* that is now called is overloaded, and thus needs an additional dictionary parameter itself.

```
 test2 =
 {store (C/Int 5); λx →
  store (C/Int 6); λy →
  call neDef dictEqInt x y
 }
```

So we still have the overhead associated with run-time dictionary passing here.

To overcome this, we rely on a technique that is actually more

general: statically apply partial evaluation and generate specialized 'clones' of a function that is called with a constant argument. Function *neDef* has three parameters, and in this call the first argument is a global constant: a dictionary, that (since the SelectMember transformation is performed) is defined as a global constant:

$$dictEqInt \leftarrow \textbf{store}\ (\mathbf{C}/Eq\ eqEqInt\ neEqInt)$$

In this particular example the other two arguments are constants as well, so the function may end up to be specialized for all three argu-

ments. In a typical situation however only the dictionary is constant, and we get a specialized copy still expecting two parameters.

For the sake of argument, we assume here that the function is only specialized for its dictionary argument (imagine an option to be active that forbids specializing for integer arguments). Starting from the original definition of *neDef*:

```
neDef d x y =
{store (F/eq d)  ; λf →
 store (A f x y); λr →
 call not r
}
```

a clone is generated, which is specialized for the first argument:

```
neDef′ x y =
  specialized (neDef (dictEqInt _ _))
{store (F/eq dictEqInt); λf →
 store (A f x y)       ; λr →
 call not r
```

}

Note that the clone is annotated in such a way that it is manifest what was the original function, and for which arguments it was specialized. This way, when the transformation is run again later, we can avoid making another clone for the same argument.

The call is adapted accordingly:

$test2 =$
$\{$**store** (**C**/Int 5); $\lambda x \rightarrow$
store (**C**/Int 6); $\lambda y \rightarrow$
call $neDef'$ x y
$\}$

5.6 ...and repeat

After the specialization, new opportunities are exposed in the body of the clone for transformations that we discussed earlier. Firstly, the operation **store** (**F**/eq $dictEqInt$) is an opportunity for the SelectMember transformation. In subsection 5.2 we described that for all selectors s that select field i from a dictionary, and all constant dictionaries d having f as it's ith field, it replaces **call** s d by **eval** f.

We now widen the task of SelectMember to also handle selections from constant dictionaries that are disguised as thunk. That is: replace **store** (**F**/s d) by **unit** f.

In this way, we lose the lazy behavior of the field selection. But there is no need for laziness in this situation: field selection cannot fail, and it is performed fast – in fact, now that we perform it statically, it takes no time at all. Nobody will oppose not postponing a call that takes zero time and cannot fail.

Another déja vu: the **unit** *eqEqInt* that emerges from the previous transformation can be combined with the thunkified **apply** in **store** $(\mathbf{A}\ f\ x\ y)$. This is done by the ApplyKnown, whose task is also widened to handle situations where an **apply** is disguised as a thunk with **A** tag.

Next, further opportunities for another SpecConst transformation may arise, etcetera. All in all, the following sequence of transformations should be performed repeatedly: SelectMember, EvalKnown, ApplyKnown, and SpecConst.

New opportunities will appear as often as overloaded functions keep calling each other, requiring as many iterations as the static nesting depth of overloaded functions. If we want to be sure that all dictionary-passing is removed, we should iterate the four transformations until we reach a fixed point. In practice, a fixed number of iterations could satisfy.

In real-life example programs involving the complicated classes from the numeric, IO, and read/show libraries, we observed the transformation of a program to converge to a fixed point after about 5 iterations.

6. Concluding remarks

The UHC compiler is written in Haskell itself. The ease of defining recursive datastructures, parsers, and tree traversals make Haskell a useful tool for compiler writing. Still, writing a tree traversal for a non-trivial language like the intermediate languages in UHC takes some effort.

This is aggravated by the fact that many transformations in UHC perform multi-pass transformations over the parse tree. The transformations in the case study of this paper follow this pattern as well: SelectMember scans the program for finding selectors, and then scans the program again for finding calls to the selectors.

To write the tree traversals for all 60 transformations in UHC, most of which are multi-pass, by hand would be too much work. That is why we employ an attribute grammar based preprocessor for Haskell, that greatly facilitates writing tree traversals [Fokker and Swierstra 2008, website UUAG]. Maybe it is even fair to say that UHC is written in the language of attribute grammars, and it is compiled into Haskell by the preprocessor. There is a vague border between the concepts of 'preprocessor' and 'compiler', between 'software generation' and 'program translation'.

6.1 Related work

The Haskell compiler that is most widely used is the Glasgow Haskell Compiler (GHC) [website GHC]. It is of production quality, and many people contribute to its implementation or to supporting libraries and applications.

Like UHC, GHC is written in Haskell itself, but without the aid of a preprocessor. GHC is structured as a sequence of transformations [Peyton Jones and Santos 1994, Santos 1995], but it tries to do as much as possible in a single transformation. This makes it harder to understand how various optimization steps cooperate and interact. On the positive side, economizing on transformations makes the compiler fast.

GHC is designed to compile modules separately, and it is interesting to see how it handles overloading. When a module is compiled,

GHC reads summaries (known as 'hi'-files) of the modules it imports. These summaries consist mainly of signatures of exported functions. But an exception is made for default definitions in class declarations: these definitions are stored in full in the summaries. In this way, GHC is able to do specialization of default defintions, at the expense of causing massive recompilation if a default definition changes, since the hi-file then changes.

There is a vast amount of literature on the implementation of overloading. We mention work on using dictionaries for implementing overloading [Augustsson 1993], dynamical merging of dictionaries in separate modules [Faxén 2002], and statical elimination of the overhead of dictionary passing [Jones 1995].

The philosophy of writing a Haskell compiler by means of many small transformations, including Grin as an intermidiate language, was proposed in the Grin project [Boquist and Johnsson 1996, Boquist 1999]. They focused on whole-program analysis for statically approximating the values of heap pointers. Overloading is not implemented in this project.

Computations over abstract syntax trees and tree transformation form a major part of any compiler. An alternative to our attribute-based approach is the Stratego system [Visser 2001,

website Stratego]. Stratego is strong in expressing local rewrites, and is able to perform transformations repeatedly. It is a large system that employs its own language. This makes it self-contained, but it lacks the ease of an underlying general purpose language.

An alternative system that does use attribute grammars for tree rewriting is Jastadd [Ekman and Hedin 2007, website Jastadd]. It

uses a different underlying language (Java). Thus, it doesn't get lazy evaluation for free, but it is easier to maintain non-local references in the abstract syntax tree.

References

[Augustsson 1993] Lennart Augustsson. Implementing Haskell overloading. In: *Functional Programming and Computer Architecture* FPCA 1993, pp 65–73.

[Bernardy et al 2009] Jean-Philippe Bernardy et al. A comparison of C++ concepts and Haskell type classes. In: ACM workshop on generic programming GP 2009, pp. 37–48.

[Boquist and Johnsson 1996] Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In: *Workshop on Implementation of Functional Languages* IFL 1996. Springer LNCS 1268.

[Boquist 1999] Urban Boquist. *Code optimisation techniques for lazy functional languages*. PhD Thesis Chalmers University, Göteborg March 1999.

[Dijkstra 2005] Atze Dijkstra. *Stepping through Haskell*. PhD Thesis Utrecht University, November 2005.

[Dijkstra, Fokker and Swierstra 2009] Atze Dijkstra, Jeroen Fokker and S. Doaitse Swierstra. The architecture of the Utrecht Haskell Compiler. In: ACM Haskell symposium Haskell'09, pp. 93–104.

[Ekman and Hedin 2007] Torbjörn Ekman and Görel Hedin. The JastAdd System: modular extensible compiler construction. *Science of Computer Programming* **69** (2007), pp. 14–26.

[Faxén 2002] Karl-Filip Faxén. A static semantics for Haskell. *J. Functional Programming* **12** (2002), pp. 295–357.

[Fokker and Swierstra 2008] Jeroen Fokker and S. Doaitse Swierstra.

Abstract interpretation of functional programs using an attribute grammar system. In: *Language Descriptions, Tools and Applications* LDTA 2008.

[Jones 1995] Mark. P. Jones. Dictionary-free overloading by partial evaluation. *Lisp and symbolic computation* **8** (1995), pp. 229–248.

[Peyton Jones 2003] Simon Peyton Jones. *Haskell 98, language and libraries: the revised report*. Cambridge univeristy press, 2003.

[Peyton Jones and Santos 1994] Simon Peyton Jones and Andre Santos. Compilation by transformation in the Glasgow Haskell Compiler. In: *Functional programming* (K. Hammond et al, ed.), pp. 184–204. Workshops in computing, Springer, 1994.

[Santos 1995] Andre Santos. *Compilation by transformation in non-strict functional languages*. PhD Thesis University of Glasgow, 1995.

[Swierstra, Azero and Saraiva 1998] S. Doaitse Swierstra, Pablo R. Azero Alocer, and João Saraiva. Designing and implementing combinator languages. In: *Advanced functional programming* AFP'98. Springer LNCS 1608.

[Visser 2001] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In: *Rewriting Techniques and Applications*, RTA'01, pp. 357–361. Springer LNCS 2051.

[website Jastadd] `www.jastadd.org`

[website Stratego] `www.program-transformation.org/Stratego`

[website GHC] `www.haskell.org/ghc`

[website UHC] `www.cs.uu.nl/wiki/bin/view/UHC`

[website UUAG] `www.cs.uu.nl/wiki/bin/view/HUT`