Strictness Optimization in a Typed Intermediate Language

Tom Lokhorst

MSc Thesis

August 26, 2010

INF/SCR-09-55

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Daily supervisor:
dr. Atze Dijkstra

Second supervisor:
prof. dr. S.D. Swierstra

Abstract

Static program analysis and compile time program optimizations are important aspects of functional language compilers. Strictness optimization is a big part of a Haskell compiler, it is interesting from a research perspective, as well as being needed for practical performance issues. In this thesis we explore a new, typed intermediate language designed for doing static program analysis. This language is well suited to implementing optimizing transformations. We implement strictness optimization for first-order functions, and see how we can safely combine this optimization with other optimizations. Finally, we discuss how we can extend the language to implement strictness optimization for higher-order functions.

Contents

Chapter 1

Introduction

High-level declarative languages like Haskell are very powerful. They allow programmers to focus on expressing their ideas in terms of *what* a program must do, rather than *how* it is done. In contrast to lower level languages like C, a Haskell programmer doesn't have to concern herself with mundane tasks like memory management or evaluation order.

All this convenience for the programmer of course comes at a price; The Haskell compiler must be very sophisticated. To have a Haskell implementation be somewhat comparable to a C compiler in terms of execution speed of the generated assembly, it must implement several static analyses and optimizations.

In this thesis, we will look at optimization techniques for the Utrecht Haskell Compiler (UHC). The compiler already has several basic optimizations, particularly in its whole-program analysis back-end. However, one glaring omission in the list of optimizations is strictness optimization.

We will explore a new intermediate language, developed for UHC. The language, called TyCore, is especially designed for doing compile time optimizations on a lazy functional language. We see how we can use the features of the language to implement optimizing transformations. One of these transformations is strictness optimization for first-order functions.

Because we haven't implemented an actual strictness analyser, we must imagine we have one. By manually annotating types in source code with so called *strictness annotations*, we have input for our strictness optimizer. The annotations are of the same form as the output of a type based program analysis would be.

After seeing how we can use TyCore to implement strictness optimization for first-order functions, we will look at higher-order functions. We propose an extension to the language allowing us to express strictness as first class properties. Using these first class strictness properties, we can better reason over strictness and optimize polyvariant functions.

In chapter 2 we look at the background of the analyses and technologies used in the rest of the thesis. Chapter 3 describes the TyCore language, and some of its defining properties. We will look at some simple optimizing transformations in chapter 4, and see how we can combine them efficiently. In chapter 5 we will see how we can apply our strictness optimization to higher-order, polyvariant functions. The finer details of the implementation of the optimizations are discussed in chapter 6.

Our primary contributions in this thesis are the following:

- We describe the TyCore language, and explain why it is an ideal language for implementing a certain type of optimizing transformations.
- We describe several optimizing transformations, as well as a novel way of using the worker/wrapper transformation to safely combine multiple transformations.
- We explore a new way for implementing strictness optimization for higher-order functions.

Chapter 2

Background

Like the whole of modern computer science, this thesis builds upon the works of others. This chapter describes the background for some of the technologies used in this thesis.

We provide a brief introduction to Type Based Program Analysis, Strict Core and the Utrecht Haskell Compiler.

2.1 Type Based Program Analysis

Static program analysis is a segment of Computer Science that has existed almost as long as the discipline itself. It has been used for two distinct purposes: verification and optimization. There are several different approaches to static program analysis, among them are: data-flow analysis, constraint based analysis, abstract interpretation, and type and effect systems [12].

Type theory is a branch of mathematics that existed since before the advent of the computer. Combined with compiled software, type theory has introduced a new and interesting field: type systems. Type systems are used to constrain the values a variable in a program can have, this allows for more verification and the generation of faster code.

A type system only describes the 'types' of values, but there are many more properties a compiler (and programmer) might want to keep track of. In type based program analysis, the type system is 'hijacked' to keep track of more properties than just types. By extending the type system into a 'type and effect system', effects can also be tracked.

An effect is a property of a value that is not directly related to its type. For example, a value might be an Integer (type) that is known at compile time (binding time effect). Or a value can be a String (type) that is guaranteed to have at most one reference to it (uniqueness effect).

The benefit of reusing the type system to also track effects is that we can also reuse all the tools, techniques and infrastructure it provides, e.g. polymorphism, subtyping and type inference.

Examples of effects in a type system include exceptions in Java [7] and uniqueness typing in Clean [18]. There is ongoing work into finding a good abstraction for type and effect systems [8] [10]. However, that work mainly focusses on the creation of a sound mathematical model, and less on actual implementation in a full (Haskell) compiler.

Throughout this document we will use the terms 'effect' and 'type annotation' interchangeably.

2.2 Strict Core

In their 2009 Haskell Symposium paper "Types Are Calling Conventions" [2] Max Bolingbroke and Simon Peyton Jones introduce the *Strict Core* language. The language is introduced as an alternative for the existing 'Core' language in the Glassgow Haskell Compiler (GHC) [17].

Strict Core is an intermediate language for purely functional languages. It is statically typed and supports lazy evaluation through expilict constructs. The language is a large part of the basis for TyCore, many of its unique features translate perfectly to TyCore.

2.3 Utrecht Haskell Compiler

Developed at Utrecht University, the Utrecht Haskell Compiler (UHC) [4] is a Haskell compiler framework. It is designed with a dual purpose in mind, to be:

- A full Haskell2010 [11] compiler with extensions
- A platform for experimentation and learning.

It has been used in several different courses, seminars and masters thesis projects at Utrecht University. In April of 2009, the first official version of the Utrecht Haskell Compiler was released [5] to the general public. This version is stable and is almost Haskell98 [13] compliant.

Since Haskell is a complex high-level language, compiling it to efficient low-level machine code can be a daunting task. To deal with this, UHC is built up from many small transformations. Each of these small transformations should be understandable due to its size. This basic design principle is made clear in the following list of 'languages' used inside the compiler. Each of these languages is a step in the compilation pipeline and itself also consist of multiple transformations.

HS The Haskell (HS) data type represents the concrete source code of a program that is parsed into an abstract syntax tree.

EH Essential Haskell is a dessugared, simplified form of Haskell. Type inference is done at this level.

Core is a representation of the source code in (a superset of) untyped lambda calculus.

GRIN (Graph Reduction Intermediate Notation) is a representation proposed by Boquist [3] specifically designed to compile lazy functional programming languages to low-level machine code.

Silly (Simple imperative little language) is an abstraction of the C language, featuring an explicit heap and stack.

C is used as the lowest level universal back-end. Primitive functions are implemented at this level.

The GRIN part of the pipeline consists of multiple transformations, even more so than the other languages. Each of these transformations, making the code more low-level. The GRIN code at the start of these transformations is so different from the GRIN code at the end of the transformations that it could be considered a different language. These two "languages" are colloquially known as "high-level GRIN" and "low-level GRIN".

UHC has multiple back-ends; A bytecode interpreter, a whole program C back-end, a LLVM back-end, and experimental Java and .NET back-ends. Note that C back-end, requires whole program analysis to do what's known as eval-inlining in GRIN. This is a fundamental part of the original GRIN language design and therefore, if this back-end is enabled, UHC has to have the whole program source available at compile time.

Recently, a new typed core language, called TyCore, has been developed. The languages is based on languages like Henk [14] and GHC's Core [17]. It encodes laziness and calling conventions explicitly in the type system, as per 'Types are calling conventions' [2] and there's thought of implementing type equality coercions [15].

The optimising transformations described in this paper occur on this TyCore language.

UHC is built using the concept of Attribute Grammars (AG). In particular, the compiler is built using the Attribute Grammar system from Utrecht University [16]. The precise details of the AG system will not be discussed in this thesis (see the original UHC book [4] for that). It suffices to say that UHC uses separate attributes to model separate aspects of the compiler. All transformations described in this thesis are implemented using Attribute Grammars.

As described before, UHC is a 'compiler framework', it isn't just a single compiler. UHC is a set of increasingly complex compilers, each one described as a delta to the previous one, building on capabilities provided by previous compilers.

These different compilers are known as 'variants' in UHC lingo. Although, there are many variants, we are only interested in three of them: The Haskell language extension for annotating types with strictness information is implemented in variant 5. The optimizing transformations are implemented in variant 8, this is the first variant where code generation is performed. The final UHC variant is 100, this variant encompasses the whole of the Utrecht Haskell Compiler.

Next to variants, UHC also uses 'aspects'. These are compile time configurable parts of the compiler. The language extension to Haskell is implemented in an aspect, such that it can be enabled at will.

Chapter 3

TyCore

TyCore is a new intermediate language in UHC's pipeline of languages. It is a combination of Henk [14], GHC core [17] and the previously mentioned Strict Core [2].

Similar in function, TyCore is designed as an alternative to the existing untyped Core language in UHC. At the time of writing, TyCore is still under development and can not be used for "real" programs. For example, Sum data types, foreign functions and module imports are not yet supported.

TyCore is a typed functional language, in that sense it is a lot like Haskell. But there are some distinctive features which set it apart from Haskell. We will discuss these difference here, trying to get an intuitive feeling for the TyCore language.

Note that, although this chapter contains a partial syntax description, it is not intended to be a *full language specification*. However, since there currently is no other formalization of TyCore, this chapter is the best specification of TyCore available at this moment.

3.1 Pipeline

To illustrate the architecture of UHC with the addition of TyCore, figure 3.1 depicts the relevant part of the pipeline. The choice for using the TyCore intermediate language can be made by means of a command line argument to the compiler.

The new pipeline behaves as follows:

- A Haskell module is loaded, parsed and simplified to Essential Haskell.
- The Essential Haskell module is type checked, this includes type inference.

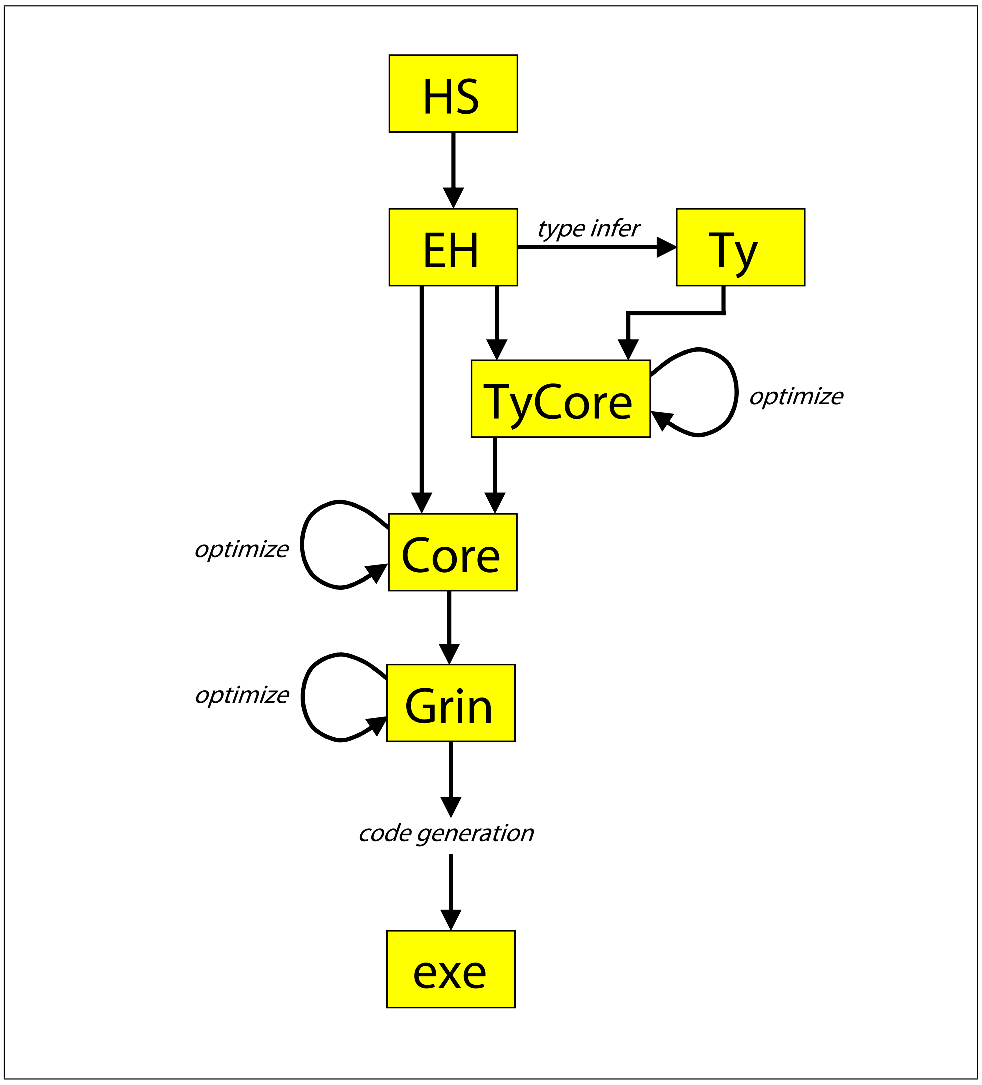


Figure 3.1: Architectural overview of TyCore intermediate language in UHC pipeline.

- At this point the module will either be transformed to Core or TyCore.
    - The Core path throws away type information and performs optimizing transformations.
    - The TyCore path keeps all type information and performs optimizing transformations which are type checked.
        - After a TyCore module is optimized, all type information is thrown away, and the module in transformed to Core.
- The Grin module performs its own optimizations, and passes the module on further down the pipeline.

3.2 Unified Term and Type Languages

TyCore is an explicitly typed intermediate language. That is, all *value* expressions in a TyCore program are explicitly annotated with a *type*. Also, all variables referring to a *type* are explicitly annotated with a *kind*. And finally, all variables referring to a *kind* are annotated with a so called *super-kind*. – Note that variable super-kinds aren't allowed in TyCore, therefor there is no need to explicitly type a super-kind in a TyCore expression.

Because of the sophisticated type information in TyCore programs, type and kind expressions can become almost as complex as value expressions. To simplify things, we use a single syntax for terms, types, and kinds. All term and type information is encoded by a single $Expr$ data type in the compiler.

The idea for a unified syntax and data type is not new, it is based the language Henk [14]. Henk is itself based on the concept of *pure type systems* (PTS), which is a generalization of the lambda cube, as described by Barendregt [1].

An obvious concern when using the same data type for terms and types is that this allows for nonsensical constructs. For example, one could attribute the type 3 to a character: "$'c' : 3$". In a system where the term and type languages are separate, this construct would not be allowed since 3 is not a type. However, even in a system with separate languages, ill-formed expressions are not prevented by the data type, e.g. applying a boolean to a number: "3 *True*". Both these examples are wrong, and while the first might have been prevented by having separate languages, the second wouldn't. To counter these ill-formed expressions, the compiler contains a type checker. This type checker validates expressions, and will report *both* types of errors.

We will look at an example to see how we use the single syntax in TyCore, in contrast to the multiple syntaxes used in other languages. This example simply wraps the identity function:

Function in polymorphic Haskell:
$\lambda x \to id\ x$

Function in second-order lambda calculus (F2):
$\Lambda\alpha.\ \lambda x : \alpha.\ id\ \langle\alpha\rangle\ x$

Function in TyCore:
$\lambda\alpha :_1 \star \to \lambda x :_0 \alpha \to id\ \alpha\ x$

Figure 3.2: Wrapping the identity function

We see that in Haskell, parametric polymorphism happens behind the scenes. The type of the function will automatically be generalized at a let binding, and the application of *id* will automatically be specialized.

In second-order lambda calculus, also called *F2*, the implicit generalization and specialization become explicit. The Λ construct introduces a type abstraction over the type α, the λ value abstraction then uses the type α again. Type application is done through the angle brackets.

In TyCore, type abstraction and application use the same syntax as value abstraction and application. However, although the λ-abstractions used are the same, the bindings are not. A binding in TyCore (encoded in syntax by a colon ":") is indexed by a level: 0 for values, 1 of types, 2 for kinds. By means of the indexing, we can see the difference between the type binding of α and the value binding x.

Note that, in contrast to Henk and PTS, TyCore does not have Π-abstraction. Instead, TyCore has an explicit function arrow ($\to$). To demonstrate the difference between the two, we use the example of typing judgements for the *K* combinator, also described in the *Henk* paper. See figure 3.3.

As discussed before, F2 uses an explicit type abstraction construct (Λ). This type abstraction is implemented as universal quantification ($\forall$) at the type level.

In contrast, PTS uses the same abstraction (λ) for both types and terms. At the type level, universal quantification is implemented using the Π-construct. E.g. $\forall\alpha.\ \beta$ in F2 becomes $\Pi\alpha : \star.\ \beta$ in PTS. The function type $\alpha \to \beta$ is also implemented using the Π-construct: $\Pi_{\_} : \alpha.\ \beta$.

Finally, TyCore combines these two syntaxes, at the term level TyCore uses λ abstraction for both terms and types like PTS. At the type level, TyCore uses only the function arrow ($\to$) for both type abstraction ($\alpha : \star \to \beta$) as

Definition of the K combinator:

$K\,x\,y \;=\; x$

Typing judgement in F2:

$\vdash (\Lambda\alpha\Lambda\beta.\ \lambda x : \alpha.\ \lambda y : \beta.\ x) : (\forall\alpha\beta.\ \alpha \to \beta \to \alpha)$

Typing judgement in PTS:

$\vdash (\lambda\alpha : \star.\ \lambda\beta : \star.\ \lambda x : \alpha.\ \lambda y : \beta.\ x) : (\Pi\alpha : \star.\ \Pi\beta : \star.\ \Pi\_ : \alpha.\ \Pi\_ : \beta.\ \alpha)$

Typing judgement in TyCore:

$\vdash (\lambda\alpha :_1 \star \to \lambda\beta :_1 \star \to \lambda x :_0 \alpha \to \lambda y :_0 \beta.\ x)$
$\quad : (\alpha :_1 \star \to \beta :_1 \star \to \alpha \to \beta \to \alpha)$

Figure 3.3: Typing the K-combinator

well as functions types ($\_ : \alpha \to \beta$). Again, the indexes on bindings allow us to see the difference between levels.

3.3 Language Definition

We will look at a subset of the language definition for TyCore. Figure 3.4 shows parts of the syntax of TyCore.

This is not a full specification of TyCore. Since the language is still under development, and not all features are in a working condition, we only focus on the parts of the language that are relevant to the contents of this thesis. In particular, we won't look at data types and **case of** expressions.

The **module** has a name and a list of bindings. Note that in this version, TyCore doesn't yet support multiple modules, so there is no facility to import other modules. A binding is of a very simple form: "$q = e_1$", here q is a binding pattern, and e_1 is an expression. We will discuss binding patterns later on (page 21).

As discussed in 3.2, the *Expr* data type is shared between terms and types. In the language definition, the symbols $e, \tau, \kappa, \varsigma$ are all used to refer to the same expression data type.

The *Expr* data type consists of 12 separate constructors. There are recursive, and non-recursive bindings (**letrec** and **let**), which each consist of a list of bindings and an expression to be evaluated with those bindings in scope. Application takes the form of "$e_1\ e_2$", where at the term level e_2 must always be a sequence of values. Functions are introduced through lambda expressions that take the form "$\lambda q \to e_1$", the q here is again a binding pattern, similar to the pattern used in **module** and **let** bindings.

Variables $x, y, \alpha, \beta, \gamma$

Module

$m ::= \textbf{module}\ x\ [b]$

Bindings

b	$::=$	$q = e_1$	Binds result of e_1 to pattern in q

Binding patterns

q	$::=$	e	Sequences with binding elements are the only valid value

Expressions

e, τ, κ, σ	$::=$	$\textbf{let}\ [b]\ \textbf{in}\ e_1$	Non-recursive bindings
	$\mid$	$\textbf{letrec}\ [b]\ \textbf{in}\ e_1$	Recusive bindings
	$\mid$	$e_1\ e_2$	Application
	$\mid$	$\lambda q \rightarrow e_1$	Lambda abstraction
	$\mid$	$\tau_1 \rightarrow \tau_2$	Function type
	$\mid$	x	Variable reference
	$\mid$	$\ell : \tau$	Integer, character or string literal
	$\mid$	$\langle \epsilon_1, \ldots, \epsilon_n \rangle$	Sequence of ϵ's
	$\mid$	$\{e\}$	`Delay` construct, introduces lazy term
	$\mid$	$\lvert e \rvert$	`Force` construct, removes laziness
	$\mid$	$\{\tau\}$	`Lazy` construct, lazy type
	$\mid$	e^{ψ}	Annotated expression

Sequence elements at value positions

ϵ	$::=$	e_0	Single term value
	$\mid$	τ_1	Single type value
	$\mid$	κ_2	Single kind value

Sequence elements at binding positions

ϵ	$::=$	$x :_0 \tau$	Term binding, annotated with type
	$\mid$	$\alpha :_1 \kappa$	Type binding, annotated with kind
	$\mid$	$\gamma :_2 \sigma$	Kind binding, annotated with super-kind

Annotations

ψ	$::=$	φ	Strictness annotation

Strictness

φ	$::=$	x	Strictness variable
	$\mid$	`S`	'Strict' or 'small' strictness property
	$\mid$	`L`	'Lazy' or 'large' strictness property

Figure 3.4: Syntax of TyCore

Function types are constructed with the function arrow "$\rightarrow$". Variables are referred by their name, e.g. "x". Integers, characters and strings can appear as literals, these literals are alway annotated with a type: "$\ell : \tau$". Sequences of elements are enclosed in angle brackets, e.g.: "$\langle \epsilon_1,\ \epsilon_2 \rangle$". Laziness at the value level is introduced and subsequently eliminated through the delay "$\{e\}$" and force "$|\ e\ |$" constructs. A lazy type is also encoded using curly braces "$\{\tau\}$", more details on laziness in TyCore later on 3.6. Finally, any expression can be annotated with an annotation "e^{ψ}". These annotations have no semantic meaning of their own, but can be used by compiler transformations to change the meaning of a program. The use of the strictness annotations is further explained in later chapters.

Sequences consists of elements that are either values or bindings. There are three different levels of value elements: values, types, and kinds. In the example "$const\ \langle Int,\ Char \rangle\ \langle 3,\ 'c' \rangle$", the first sequence $\langle Int,\ Char \rangle$ is a sequence of types, the second sequence $\langle 3,\ 'c' \rangle$ is a sequence of values. In the example this is apparent from the context, but in the actual *Expr* data type there are three different constructors for each level.

A sequence of binding elements is called a *binding pattern*. There are three levels of binding elements: bindings for values, types and kinds. The following example shows a sequence of type bindings and a sequence of value bindings: "$\lambda \langle \alpha :_1 \star,\ \beta :_1 \star \rangle \rightarrow \lambda \langle x :_0 \alpha,\ y :_0 \beta \rangle \rightarrow x$".

3.4 Syntactic sugar

Because a TyCore expression can consist of many different constructors, we sometimes use syntactic sugar to simplify an expression for readability. Figure 3.5 shows a few shorthands used, as well as their expansions.

Shorthand		Expansion	
e	$\triangleq$	$\langle e \rangle$	Singleton values
τ	$\triangleq$	$\langle \tau \rangle$	Singleton types
$x : \tau$	$\triangleq$	$x :_0 \tau$	Value binding
$\lambda \langle x : \tau_1 \rangle\ \langle y : \tau_2 \rangle \rightarrow e$	$\triangleq$	$\lambda \langle x : \tau_1 \rangle \rightarrow \lambda \langle y : \tau_2 \rangle \rightarrow e$	Curried function
$\{\tau_1, \ldots, \tau_n\}$	$\triangleq$	$\{\langle \tau_1, \ldots, \tau_n \rangle\}$	Lazy sequences

Figure 3.5: Overview of syntactic sugar

The following example shows the difference between a syntactic shorthand version, and the fully expanded version:

$$
\begin{aligned}
& id : \{Int\} \rightarrow Int \\
& \quad = \lambda x : \{Int\} \rightarrow \\
& \qquad |x|
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{id} : \langle\{\langle \mathit{Int}\rangle\}\rangle \to \langle \mathit{Int}\rangle\\
&\quad = \lambda\langle x : \{\langle \mathit{Int}\rangle\}\rangle \to\\
&\qquad \langle |x| \rangle
\end{aligned}
$$

This example shows the shorthand form of a singleton value sequence, a singleton type sequence and a lazy sequence.

3.5 Explicit Types

One of the main differentiating features between TyCore and the existing Core language in UHC are types. TyCore is statically and strongly typed, which brings two main advantages;

- Transformations that operate on a TyCore AST (like the optimizing transformations described later in this thesis), can be checked to produce type correct ASTs. This is a powerful feature that adds an extra layer of safety. For example, it is quite possible to accidentally produce an expression like "3 *even*" as the result of a transformation. I.e. apply a function to an argument instead of the other way around. This produces a valid AST, therefor this error won't be detected by GHC's type checker. However this is a type incorrect TyCore program and the error will be discovered by the TyCore type checker.
- The code generator can use types to produce more efficient code. This is of course the main point in the paper "Types are Calling Conventions" [2]. TyCore's features discussed below (explicit laziness and multiple arguments/results) allow for expressive types that can be translated to efficient code, relatively straightforward.

TyCore is explicitly typed; All types that need to be known at compile type by the type checker are explicitly provided, and therefor don't need to be inferred. Types are specified at two locations in the the AST; All literals (integers, characters and strings, currently) are annotated with their type, and name bindings are annotated with the correct type.

Here are two examples of both forms of type annotations:

-- Literals:
$\mathit{const}'\ \langle 3 : \mathit{Int}\rangle\ \langle \texttt{'c'} : \mathit{Char}\rangle$

-- Name bindings:

$$
\begin{aligned}
&\mathit{const}' : \langle \mathit{Int}\rangle \to \langle \mathit{Char}\rangle \to \langle \mathit{Int}\rangle\\
&\quad = \lambda\langle x : \mathit{Int}\rangle\ \langle y : \mathit{Char}\rangle \to \langle x\rangle
\end{aligned}
$$

In contrast to Haskell, TyCore does not have separate type signatures. A binding pattern always consists of a name and a type. The following **let** expression demonstrates this.

```
    -- Let bindings:
$id' : \langle Int \rangle \rightarrow \langle Int \rangle$
    $= \lambda \langle x : Int \rangle \rightarrow$
        **let** $c : \langle Char \rangle$
                $=$ 'c'
            $const' : \langle Int, Char \rangle \rightarrow \langle Int \rangle$
                $= \lambda \langle x : Int, \_ : Char \rangle \rightarrow \langle x \rangle$
        **in** $const'\ \langle x, c \rangle$
```

Each binding of the form "$q = e$" consists of a left hand side which is a sequence of names and types, and a right hand side which is an expression.

3.6 Strict Semantics

In contrast to Haskell, TyCore is a strict language, i.e. arguments to a function call are to be evaluated before the function call itself.

We will demonstrate the difference in evaluation order by means of an example:

$$\begin{aligned}
&foo : Int \rightarrow Int \rightarrow Int\\
&\quad = \lambda x\ y \rightarrow succ\ x\\
&succ : Int \rightarrow Int\\
&\quad = \lambda x \rightarrow x + 1
\end{aligned}$$

In a lazy language, evaluation proceeds as follows:

$$\begin{aligned}
&foo\ (2 \star 3)\ (1 + 2)\\
&\equiv \quad \text{-- definition of 'foo'}\\
&succ\ (2 \star 3)\\
&\equiv \quad \text{-- definition of 'succ'}\\
&(2 \star 3) + 1\\
&\equiv \quad \text{-- evaluate arithmetic expression}\\
&7
\end{aligned}$$

But in a strict language, both arguments to the 'const' functions will be evaluated (in left to right order), before entering the function.

$foo\ (2 \star 3)\ (1+2)$
$\equiv$ -- evaluate arguments
$foo\ 6\ 3$
$\equiv$ -- definition of 'foo'
$succ\ 6$
$\equiv$ -- definition of 'succ'
$6+1$
$\equiv$ -- evaluate arithmetic expression
7

Also note that in a strict language like TyCore, the call "$foo\ (1+2)\ (4\ /\ 0)$" will diverge, whereas it would not in a lazy language.

3.6.1 Restoring laziness

Although TyCore is a strict language, it can support Haskell's lazy semantics through explicit lazy constructs. Laziness is not only supported at the term level, but also in the type system. The type of a strict integer and the type of a lazy integer are different.

By making laziness explicit in a strict language, we can reason about evaluation order and create optimizing transformations that increase runtime efficiency of a program by changing where laziness is introduced and removed.

As a side note, others [9] have used two types of function application to support strictness in a Haskell-like language. By means of strict function application ($!) and lazy function application ($) the correct semantics can be encoded. While this is an attractive solution, it is not usable in TyCore. Since functions take multiple arguments, and the strictness of each of the arguments must be specified, a single distinction at the application-site is not enough.

Delay

The *Delay* construct in TyCore delays a computation and creates a thunk instead. Delayed computations are enclosed in curly braces in the concrete syntax of TyCore:

-- Strict addition:
$x : Int$
$\quad = 1 + 2$

-- Lazy addition:
$y : \{Int\}$
$\quad = \{1+2\}$

Force

Suspended computations, or thunks, that are introduced by the *Delay* construct, can be evaluated by means of the *Force* construct. In syntax this is encoded by enclosing an expression between two vertical bars:

$z : Int$
$\quad = |y|$

The operational semantics of *Force* are the same as those in Haskell. If a thunk is evaluated, it is updated by its value. Forcing a thunk for the second time will reuse the already computed value.

3.7 Multiple Arguments & Results

All functions in TyCore take a sequence of arguments and return a sequence of results. These can be singleton sequences, but they are never single values. Note that for readability we sometimes drop the angle brackets around singleton sequences, especially around top level name bindings, but this is just syntactic sugar.

The following examples demonstrate sequences in argument and result positions.

$const : \langle Int, Char\rangle \to \langle Int\rangle$
$\quad = \lambda\langle x : Int, c : Char\rangle \to \langle x\rangle$
$dup : \langle Int\rangle \to \langle Int, Int\rangle$
$\quad = \lambda\langle x : Int\rangle \to \langle x, x\rangle$

It is important to remember that sequences are different from tuples in Haskell. E.g. a Haskell programmer might be tempted to write:

$foo : \langle Int, Char\rangle \to \langle Int\rangle$
$\quad = \lambda y : \langle Int, Char\rangle \to const\ \langle y\rangle$

However this is not valid TyCore, a sequence (in the type) must always be matched with a correct sequence binding. To make this more intuitive, consider this C function:

```
int power(int base, int exp)
{
    return exp == 1 ? base : base * power(base, exp -1);
}
```

The *power* function takes a sequence of two arguments (*base* and *exp*), these cannot be match simultaneously in a single binding.

Of course, we can "assign" multiple values to a single variable by introducing explicit laziness; We can create a thunk of a sequence. For example:

$$\begin{aligned}&\mathit{bar} : \{\langle \mathit{Int}, \mathit{Int}\rangle\}\\&\quad = \{\mathit{dup}\ \langle 3\rangle\}\end{aligned}$$

Which we can subsequently force, to again get the sequence back.

$$\begin{aligned}&\langle x : \mathit{Int}, y : \mathit{Int}\rangle\\&\quad = |\mathit{bar}|\end{aligned}$$

3.8 A common pattern

An interesting pattern pattern that arises in TyCore programs that originate from Haskell is a `Delay-Force` expression. For example, the following Haskell program:

$$\begin{aligned}&\mathit{foo} :: \mathit{Int} \to \mathit{Int}\\&\mathit{foo}\ x = \mathit{const}\ x\ 3\end{aligned}$$

Will generate the following TyCore program:

$$\begin{aligned}&\mathit{foo} : \langle\{\mathit{Int}\}\rangle \to \langle \mathit{Int}\rangle\\&\quad = \lambda\langle x : \{\mathit{Int}\}\rangle \to \mathit{const}\ \langle\{|x|\}\rangle\ \langle\{3\}\rangle\end{aligned}$$

Here, the argument to the *const* function is first delayed, and then forced: $\{|x|\}$. This operation apparently achieves nothing, but the reason this code gets generated is interesting.

Because Haskell is a lazy language, all arguments to functions are lazy, i.e. the arguments to *const* must be a suspended computations. Whenever a Haskell program does a function call, we delay all arguments: $\mathit{const}\ \langle\{...\}\rangle\ \langle\{3\}\rangle$.

Since all arguments to functions are lazy, so is x, the argument too *foo*. Therefor, whenever we want to use a parameter of a function in its body, we must first `Force` it: $|x|$.

A general rule of thumb for TyCore programs that originate from Haskell is the following:

- At all function calls, all arguments to the function are `Delay`ed.
- All function parameters themselves have `Lazy` types.
- At all use-sites of function parameters inside a function body, the parameter must first be `Force`d.

To sum up: While the `Delay-Force` construct is an apparent no-op, its origins are interesting. Also, when doing transformations, the pattern is sometimes useful, we therefor leave it in the program as long as possible.

Chapter 4

Combining Optimizations

A common complication when implementing compiler optimizations is the fact that different optimizations influence each other. For simplicity's sake, the developer of an optimizing transformation would like to think of his transformation in isolation. However in practice, a single optimization is almost never run by itself.

Another difficulty when implementing optimizations is the fact that the compilation process is often modular. That is, due to separate compilation, only one module can be optimized at a time. While UHC does have a whole program back-end, this is not always enabled. Also, the whole-program optimization only happens at the level of GRIN, the optimizations described in this chapter work on TyCore.

This chapter describes our approach to implementing multiple optimizing transformations in such a way that they are relatively independent. Using our technique, we can implement transformations that do not know of each other. These transformations can be run in any order, the result is always the most optimal.

4.1 Two optimizations

We start of by looking at two simple optimizations. For the moment we will ignore modularity and the fact that they will have to be combined.

Our description of the transformation is the most straightforward and naive.

4.1.1 Definition-site arity raising

This optimizing transformation is based on a similar transformation on Strict Core in section 6.1 of the paper "Types are Calling Conventions" [2].

Multiple arguments

Consider a Haskell function called f that takes two arguments, and then calculates a result. In Haskell, however, a function can only have a single argument. The solution is to implement the function f as a curried function, once it receives its first argument, it returns a new closure that takes the second argument. The returned function captures the value of the first argument in its environment.

$$\begin{array}{l} f :: Int \rightarrow Int \rightarrow Int \\ f\ x\ y = ... \end{array}$$

$$foo = f\ 2\ 3$$

A simple translation of this Haskell program results in the following TyCore program:

$$\begin{array}{l} f : \langle Int \rangle \rightarrow \langle Int \rangle \rightarrow \langle Int \rangle \\ \quad = \lambda \langle x : Int \rangle \rightarrow \lambda \langle y : Int \rangle \rightarrow ... \end{array}$$

$$\begin{array}{l} foo : Int \\ \quad = f\ \langle 2 \rangle\ \langle 3 \rangle \end{array}$$

While this is a perfectly valid TyCore program, it is not the most optimal. Although we haven't formally specified operational semantics for TyCore, we can informally say that curried functions are always implemented as closures. However, since TyCore supports functions with multiple arguments, we can pass both arguments to f at the same time, thus removing the need for creating a closure.

$$\begin{array}{l} f' : \langle Int, Int \rangle \rightarrow \langle Int \rangle \\ \quad = \lambda \langle x : Int, y : Int \rangle \rightarrow ... \end{array}$$

$$\begin{array}{l} foo : Int \\ \quad = f'\ \langle 2, 3 \rangle \end{array}$$

This version of the program is more optimal at both the call site (only a single application is needed), as well as at the definition site (no need to create an additional closure).

Partial application

A benefit of curried functions in Haskell is that we can easily implement partial applications. In the following example, only the first argument to *f* is supplied in the definition of *bar*:

$$\begin{aligned}
&f :: Int \rightarrow Int \rightarrow Int\\
&f\ x\ y = ...\\
&\\
&bar = f\ 2
\end{aligned}$$

When we translate this program to TyCore, we also want to retain the ability to do partial applications. This is easy to implement by simply re-wrapping the call to the optimized f' in a new lambda:

$$\begin{aligned}
&f' : \langle Int, Int\rangle \rightarrow \langle Int\rangle\\
&\quad = \lambda\langle x : Int, y : Int\rangle \rightarrow ...\\
&\\
&bar : \langle Int\rangle \rightarrow \langle Int\rangle\\
&\quad = \lambda\langle y : Int\rangle \rightarrow f'\ \langle 2, y\rangle
\end{aligned}$$

Sharing

Note that we can't always apply this transformation in a type directed way. That is, simply by looking at the type $\langle Int, Char\rangle \rightarrow \langle Int\rangle \rightarrow \langle Int\rangle$, we might think we can optimize this function to one taking three simultaneous arguments, but this depends on the definition of the function. For example, take this function *g*:

$$\begin{aligned}
&g : \langle Int, Char\rangle \rightarrow \langle Int\rangle \rightarrow \langle Int\rangle\\
&\quad = \lambda\langle x : Int, c : Char\rangle \rightarrow\\
&\qquad \mathbf{let}\ z : Int = fibonacci\ \langle x\rangle\ \mathbf{in}\\
&\qquad \lambda\langle y : Int\rangle \rightarrow ...\\
&\\
&foo : \langle Int\rangle \rightarrow \langle Int\rangle\\
&\quad = g\ \langle 23, \texttt{'c'}\rangle\\
&\\
&bar : Int\\
&\quad = foo\ \langle 2\rangle\\
&\\
&baz : Int\\
&\quad = foo\ \langle 3\rangle
\end{aligned}$$

The result of computing *fibonacci* $\langle x \rangle$ is stored in the closure returned by *g*. So when *bar* and *baz* refer to *foo*, they share the already computed value.

If the *g* function were changed to take 3 arguments simultaneously, the computed *z* would not be shared between the two different invocations of *foo*.

For this reason we base the "definition site arity raising"-transformation on the body of a function (instead of its type). We only remove closures that don't have a **let** binding in between. We do apply this transformation recursively though, e.g. the following program:

$$
\begin{aligned}
& h : \langle Int \rangle \to \langle Char \rangle \to \langle Char \rangle \to \langle Int \rangle \to \langle Int \rangle \\
& \quad = \lambda \langle x : Int \rangle \to \lambda \langle c : Char \rangle \to \\
& \qquad \mathbf{let} \ldots \mathbf{in} \\
& \qquad \lambda \langle d : Char \rangle \to \lambda \langle y : Int \rangle \to \ldots
\end{aligned}
$$

is transformed to:

$$
\begin{aligned}
& h' : \langle Int, Char \rangle \to \langle Char, Int \rangle \to \langle Int \rangle \\
& \quad = \lambda \langle x : Int, c : Char \rangle \to \\
& \qquad \mathbf{let} \ldots \mathbf{in} \\
& \qquad \lambda \langle d : Char, y : Int \rangle \to \ldots
\end{aligned}
$$

4.1.2 First-order strictness optimization

Strictness optimization for first-order values is relatively straightforward. If a function has lazy parameters that are annotated to be strict, the laziness will be removed.

Note that we use the explicit type annotations provided by end users through our Haskell language extension. In TyCore, these annotations have the form: $\{@strict\ \tau\}$.

The const function

In Haskell, all arguments to a function are passed lazily. In TyCore we can make the distinction between evaluated and delayed arguments. We will look at a monomorphic *const* function in Haskell:

$$
\begin{aligned}
& const :: Int \to Int \to Int \\
& const\ x\ y = x
\end{aligned}
$$

$$foo = const\ (4 + 3)\ (fibonacci\ 23)$$

Below is this function translated to TyCore. Note that no arity raising is performed, although we do use the syntactic sugar described in 3.4.

Also, we see the first parameter to *const* is annotated with the *@strict* annotation, this has been done manually.

$$\begin{aligned}&const : \langle\{@strict\ Int\}\rangle \rightarrow \langle\{Int\}\rangle \rightarrow \langle Int\rangle\\ &\quad = \lambda\langle x : \{@strict\ Int\}\rangle\ \langle c : \{Int\}\rangle \rightarrow \langle|x|\rangle\end{aligned}$$

$$foo = const\ \langle\{4+3\}\rangle\ \langle\{fibonacci\ 23\}\rangle$$

The strictness optimization is simple; For all parameters that are lazy ($\{\tau\}$) and annotated with *@strict*, we remove the laziness. Also, inside the body of the function, we remove the force construct ($|e|$), and of course at the call site, we remove the delay construct ($\{e\}$):

$$\begin{aligned}&const' : \langle Int\rangle \rightarrow \langle\{Int\}\rangle \rightarrow \langle Int\rangle\\ &\quad = \lambda\langle x : Int\rangle\ \langle c : \{Int\}\rangle \rightarrow \langle x\rangle\end{aligned}$$

$$foo = const'\ \langle 4+3\rangle\ \langle\{fibonacci\ 23\}\rangle$$

Lazy sequences

Since TyCore allows functions to have sequences of multiple parameters, a strict-annotated lazy parameter can be part of a sequence. Or a lazy parameter can be a sequence of strict-annotated values:

$$\begin{aligned}&bar : \langle\{@strict\ Char\},\{Int\}\rangle \rightarrow \langle Char\rangle\\ &\quad = \lambda\langle c : \{@strict\ Char\}, z : \{Int\}\rangle \rightarrow \langle|c|\rangle\end{aligned}$$

$$\begin{aligned}&baz : \langle\{@strict\ Int, @strict\ Char\}\rangle \rightarrow \langle Int\rangle\\ &\quad = \lambda\langle x : \{@strict\ Int, @strict\ Char\}\rangle \rightarrow\\ &\qquad\quad \mathbf{let}\ \langle y : Int, c : Char\rangle = |x|\\ &\qquad\quad \mathbf{in}\ \langle y\rangle\end{aligned}$$

Both *bar* and *baz* can be optimized:

$$\begin{aligned}&bar' : \langle Char, \{Int\}\rangle \rightarrow \langle Char\rangle\\ &\quad = \lambda\langle c : Char, z : \{Int\}\rangle \rightarrow \langle|c|\rangle\end{aligned}$$

$$\begin{aligned}&baz' : \langle Int, Char\rangle \rightarrow \langle Int\rangle\\ &\quad = \lambda\langle x_0 : Int, x_1\ Char\rangle \rightarrow\\ &\qquad\quad \mathbf{let}\ \langle y : Int, c : Char\rangle = \langle x_0, x_1\rangle\\ &\qquad\quad \mathbf{in}\ \langle y\rangle\end{aligned}$$

When removing the laziness of sequences, multiple new bindings are introduced (e.g. x_0 and x_1). Also, all uses of a forced sequence are updated. Another transformation might later on remove the unnecessary introduction of y and c in baz'.

4.2 Modularity

Like GHC, UHC uses separate compilation, that is each module of a Haskell program is compiled separately and the compiled modules are only linked together at the very end of compilation.

Essentially, each module goes through four steps; Parsing of Haskell source, Essential Haskell type checking, (Ty)Core optimizations, GRIN byte code generation.

After the generation of GRIN bytecode there are two possible back-ends, by default the GRIN bytecode modules are loaded into memory and are simply interpreted. Alternatively, the GRIN bytecode modules are glued together and "whole program" optimized. This doesn't negate the fact that almost the entire pipeline UHC uses separate compilation, in particular all TyCore transformations happen on a per module level.

4.2.1 Worker/Wrapper separation

An obvious problem that occurs when doing optimizing transformations on a per-module level, is that the other modules won't be aware of this, as they are not part of that compilation process.

Lets look at an example of a program consisting of two modules *Lib* and *Prog*. The module *Lib* defines the *const* function, the module *Prog* imports the other module and uses the *const* function:

$$\begin{aligned}
&\textbf{module } \mathit{Lib}\\
&\mathit{const} : \langle \mathit{Int}\rangle \to \langle \mathit{Int}\rangle \to \mathit{Int}\\
&\quad = \lambda\langle x : \mathit{Int}\rangle\ \langle y : \mathit{Int}\rangle \to x
\end{aligned}$$

$$\begin{aligned}
&\textbf{module } \mathit{Prog}\\
&\textbf{import } \mathit{Lib}\\
&\mathit{foo} : \mathit{Int}\\
&\quad = \mathit{const}\ \langle 2\rangle\ \langle 3\rangle
\end{aligned}$$

As we've seen, the *const* function can be optimized by transforming it into a function that takes two arguments simultaneously.

However, the *Prog* module is not being compiled when optimizing the *Lib* module (it might not even exist at that point). Therefor, the transformation won't be able to update the call-site of *const* in *Prog*.

The solution to this problem is of course the Worker/Wrapper transformation [6]. By creating a new function that does all the work, called the worker. And wrapping that function with a function that has the same name and type signature as the original, the *Prog* module can still refer to the original *const*:

module *Lib*

$const_{worker} : \langle Int, Int \rangle \rightarrow Int$
$\quad = \lambda \langle x : Int, y : Int \rangle \rightarrow x$

$const : \langle Int \rangle \rightarrow \langle Int \rangle \rightarrow Int$
$\quad = \lambda \langle x : Int \rangle \ \langle y : Int \rangle \rightarrow const_{worker} \ \langle x, y \rangle$

module *Prog*

import *Lib*

$foo : Int$
$\quad = const \ \langle 2 \rangle \ \langle 3 \rangle$

Here we see that the call to *const* from the *Prog* module is still valid, as the optimized *Lib* module now consists of two bindings; The wrapper (*const*) with the original name and type signature, and the optimized worker function ($const_{worker}$).

Unfortunately, we've probably just decreased performance by introducing a level of indirection. We remedy this later on in the compilation pipeline. There is a point where both modules are known by the compiler, when they are merged into the single executable, this process is traditionally known as "linking".

By making the linker slightly more intelligent, we can remove the introduced indirection. We start by inlining all wrappers at their call-sites:

module *Prog*

import *Lib*

$foo : Int$
$\quad = (\lambda \langle x : Int \rangle \ \langle y : Int \rangle \rightarrow const_{worker} \ \langle x, y \rangle) \ \langle 2 \rangle \ \langle 3 \rangle$

Inlining can be a difficult transformation in compiler optimization, because there is usually a trade-off between increased performance and increased code size. In this case though, it is always safe to inline the wrapper, as the only thing it does is call the worker.

After inlining the body of the wrapper, we beta-reduce the new expression to remove the λ-expression and applications:

$$
\begin{array}{l}
\textbf{module}\ \mathit{Prog} \\
\textbf{import}\ \mathit{Lib} \\
\mathit{foo} : \mathit{Int} \\
\quad = \mathit{const}_{\mathit{worker}}\ \langle 2, 3 \rangle
\end{array}
$$

In conclusion, our modular design for optimizing transformations is a two step process; Introduction of worker/wrapper when optimizing individual modules, and inlining wrappers when linking optimized modules together.

4.3 Combining transformations using worker/wrapper

In the previous sections, we've seen two separate optimizing transformations as well as a technique for optimizing libraries in a modular way. In this section we combine these into a single concise method for implementing optimizing transformations for TyCore.

The aforementioned transformations are implemented in UHC using the technique described in this chapter. We can enable either of the two transformations, and independent of the order, will always arrive at the most optimal solution.

4.3.1 A common subset

Obviously, both the "definition site arity raising" transformation, as well as the "first-order strictness optimization" transformation, drastically change both the type and body of a function. We would like to design both transformations in such a way that they don't know of each others existence.

Our solution is to make both transformations familiar with the worker/wrapper model. All transformations on TyCore need to be familiar with, and respect the invariants of, the worker/wrapper model. In return, the transformations don't need to be aware of other transformations, thus they can be designed independently.

4.3.2 A running example

For the purposes of this section we will use the following example:

$$\begin{aligned}&\mathit{const} : \langle\{\text{@}\mathit{strict}\ \mathit{Int}\}\rangle \to \langle\{\mathit{Char}\}\rangle \to \langle\mathit{Int}\rangle\\ &\quad = \lambda\langle x : \{\text{@}\mathit{strict}\ \mathit{Int}\}\rangle\ \langle y : \{\mathit{Char}\}\rangle \to \langle|x|\rangle\end{aligned}$$

We see that both the arity raising, as well as the strictness optimization transformations can be applied. Using the worker wrapper model, the final output should be:

$$\begin{aligned}&\mathit{const}_{\mathit{worker}} : \langle\mathit{Int}, \{\mathit{Int}\}\rangle \to \mathit{Int}\\ &\quad = \lambda\langle x : \mathit{Int}, y : \{\mathit{Int}\}\rangle \to \langle x\rangle\end{aligned}$$

$$\begin{aligned}&\mathit{const} : \langle\{\mathit{Int}\}\rangle \to \langle\{\mathit{Char}\}\rangle \to \langle\mathit{Int}\rangle\\ &\quad = \lambda\langle x : \{\mathit{Int}\}\rangle\ \langle y : \{\mathit{Char}\}\rangle \to\\ &\qquad\quad \mathbf{let}\ x_0 = |x|\ \mathbf{in}\\ &\qquad\quad \mathit{const}_{\mathit{worker}}\ \langle x_0, y\rangle\end{aligned}$$

4.3.3 Four situations

When transforming a function, one of four situations can occur:

- **Ignored** The current function is not a candidate for optimization. This function will be ignored by the rest of the transformation.
- **Introduction** The current function is a 'normal' function, and is a candidate for optimization. The function will be removed and a new worker and wrapper will be introduced.
- **UpdateWorker** The current function is an existing worker, and is a candidate for optimization. The worker will be updated with a new type and definition.
- **UpdateWrapper** The current function is an existing wrapper. If the related worker is a candidate for optimization, the wrapper will be updated to correctly call the worker.

A 'normal' function in this context is a function that is neither a worker nor a wrapper.

Ignored

This is the most straightforward situation, when a function can't be optimized, it is ignored by the transformation. This applies to both 'normal' functions as well as workers and wrappers generated by previous transformations.

Introduction

When a function is a candidate for optimization and no previous transformation has touched it, the original function is removed and a new worker and wrapper are introduced.

This is a two step process; First the worker and wrapper are generated:

The worker is generated by simply copying the original function and renaming the binding (i.e. adding the $_{worker}$ suffix). The wrapper is generated by creating a new function with the same name and type as the original function, all it does is call the worker (which at this point is still a copy of the original).

In our example, the following will be generated:

$$\begin{aligned} &const_{worker} : \langle\{@strict\ Int\}\rangle \rightarrow \langle\{Char\}\rangle \rightarrow \langle Int\rangle \\ &\quad = \lambda\langle x : \{@strict\ Int\}\rangle\ \langle y : \{Char\}\rangle \rightarrow \langle|x|\rangle \end{aligned}$$

$$\begin{aligned} &const : \langle\{@strict\ Int\}\rangle \rightarrow \langle\{Char\}\rangle \rightarrow \langle Int\rangle \\ &\quad = \lambda\langle x : \{@strict\ Int\}\rangle\ \langle y : \{Char\}\rangle \rightarrow const_{worker}\ \langle\{|x|\}\rangle\ \langle\{|y|\}\rangle \end{aligned}$$

Secondly, after the dummy worker and wrapper functions are generated, the code for *UpdateWorker* and *UpdateWrapper* is called. These two pieces of code will correctly update the worker to the more efficient function and the wrapper to correctly call the updated worker.

UpdateWorker

This is the situation where the function that is being transformed is a worker, that is already introduced at an earlier stage. Lets look at an example where the strictness optimization has already occurred, and we're now running the arity raising transformation:

$$\begin{aligned} &const_{worker} : \langle Int\rangle \rightarrow \langle\{Char\}\rangle \rightarrow \langle Int\rangle \\ &\quad = \lambda(\langle x : Int\rangle\ \langle y : \{Char\}\rangle \rightarrow \langle x\rangle \end{aligned}$$

The worker can be changed in any way the transformation sees fit. In this case the type and the definition will be changed; The first two parameters will be merged:

$$\begin{aligned} &const_{worker} : \langle Int, \{Char\}\rangle \rightarrow \langle Int\rangle \\ &\quad = \lambda\langle x : Int, y : \{Char\}\rangle \rightarrow \langle x\rangle \end{aligned}$$

The accompanying wrapper will of course have to be updated to match this new type signature. This is described in the next situation.

UpdateWrapper

After the worker is transformed, the wrapper will have to be updated to match the new type signature. Lets again look at an example where the strictness optimization has already occurred, and we're now running the arity raising transformation:

$$\begin{aligned}
&\mathit{const} : \langle\{\mathit{Int}\}\rangle \rightarrow \langle\{\mathit{Char}\}\rangle \rightarrow \langle\mathit{Int}\rangle\\
&\quad = \lambda\langle x : \{\mathit{Int}\}\rangle\ \langle y : \{\mathit{Char}\}\rangle \rightarrow\\
&\qquad\quad \textbf{let}\ \langle x_0 : \mathit{Int}\rangle = |x|\ \textbf{in}\\
&\qquad\quad \mathit{const}_{\mathit{worker}}\ \langle x_0\rangle\ \langle\{|y|\}\rangle
\end{aligned}$$

We see that the strictness optimization has introduced a **let**-binding in the body of the wrapper. In the type signature and the function the *@strict* annotation has been removed, but since annotations don't impact the actual type, this is a safe change.

We will now update the call to the worker:

$$\begin{aligned}
&\mathit{const} : \langle\{\mathit{Int}\}\rangle \rightarrow \langle\{\mathit{Char}\}\rangle \rightarrow \langle\mathit{Int}\rangle\\
&\quad = \lambda\langle x : \{\mathit{Int}\}\rangle\ \langle y : \{\mathit{Char}\}\rangle \rightarrow\\
&\qquad\quad \textbf{let}\ \langle x_0 : \mathit{Int}\rangle = |x|\ \textbf{in}\\
&\qquad\quad \mathit{const}_{\mathit{worker}}\ \langle x_0, \{|y|\}\rangle
\end{aligned}$$

The wrapper is now updated to work correctly with the arity of the updated worker. Note that this transformation must be careful to not change the type signature of the original function. It must also respect **let**-bindings introduced by earlier transformations.

4.3.4 Inlining the wrapper

To wrap up the transformation process, the final stage is the inlining of the wrapper function at the call site. This is described in the Modularity section 4.2.

Since this is common to all transformations, this is can be implemented once, and doesn't need to be implemented by the transformation author.

Chapter 5

Polyvariant Strictness Optimization

In this chapter we explore a theoretical method for performing strictness optimization for higher-order functions. We start by looking at first-order functions, then we look at the possible strictness annotations, a TyCore representation for these annotations and finally code generation.

5.1 First-Order Functions

In first order functions, strictness analysis is relatively straightforward. If a function takes an argument, and the argument is guaranteed to be used in the production of the result of that function, the function is strict in that argument.

Or, more formally: if a function diverges when it gets a diverging argument, then the function is strict in that argument:

$$f\ \bot \leadsto \bot$$

Note that $\bot$ represents a diverging computation, an error, or non-termination. In Haskell this is usually called `undefined`.

We'll look at two example functions, *succ* and *ite*. The *succ* function computes the successor of its argument. The *ite* function takes a *Bool* argument and two arguments of type *a*. If the first argument evaluates to *True*, the function returns its second argument, otherwise, the third argument will be returned.

$$\begin{aligned}&succ :: Int \to Int\\&succ\ x = x + 1\end{aligned}$$

$$\begin{array}{l} ite :: Bool \to a \to a \to a \\ ite\ True\ \ x\ \_ = x \\ ite\ False\ \_\ y = y \end{array}$$

The following examples demonstrate the strictness properties of these two functions:

$$\begin{array}{llll} succ\ \bot & & & \leadsto \bot \\ ite & True\ 0\ \bot & & \not\leadsto \bot \\ ite & \bot \quad 0\ 1 & & \leadsto \bot \end{array}$$

The *succ* function is strict in its first argument, because it *always* uses it to compute its result. The *ite* function on the other hand, is not strict in its second or third argument, because neither are *always* used. It is, however, strict in its first argument, because the boolean value is always inspected to determine which of the last two arguments to return.

In type based program analysis, we use the type system to both do the actual analysis, as well as a mechanism to encode the results of the analysis. In particular, in strictness analysis, we annotate the arguments of a function with S or L, to represent strict and non-strict parameters, respectively.

In our examples, a type based strictness analysis, will compute the following types:

$$\begin{array}{ll} succ & :: Int^S \to Int \\ ite & :: Bool^S \to a^L \to a^L \to a \end{array}$$

5.2 Higher-Order Functions

Strictness analysis becomes more complicated when dealing with higher-order functions, in particular, functions that take a function as an argument. Lets look at the *app* function as an example:

$$\begin{array}{l} app :: (a \to b) \to a \to b \\ app\ f\ x = f\ x \end{array}$$

The *app* function, also known as the function-application function, takes a function f of type $a \to b$ as its first argument, and a value x of type a as its second argument. To compute its result, *app* applies x to f. Lets try to annotate this function with its strictness properties:

$$app :: (a \to b)^S \to a^? \to b$$

The function is obviously strict in its first argument (the function f), but is it also strict in its second argument (x)?

One way of reasoning tells us that the x is *used* here, since it is passed as an argument to f. But if we assume x is strict, then the following should be true:

$$app\ (\lambda\_ \rightarrow 6)\ \bot \rightsquigarrow \bot \quad \text{-- This doesn't hold!}$$

This is obviously not true. In a lazy language like Haskell, this will compute 6, it should not diverge!

An alternative is that we choose x to be L. While this is always a safe choice, is it also an optimal choice? Lets look at two uses of *app*:

$$\begin{aligned}&foo = ite\ True\ 1\\ &app\ foo\ \ \ \bot \rightsquigarrow 1\\ &app\ succ\ \bot \rightsquigarrow \bot\end{aligned}$$

We see that x is sometimes evaluated, but we can also see exactly *when* it is evaluated, let us look at the strictness properties of the two function arguments to *app*:

$$\begin{aligned}&foo\ \ :: Int^{L} \rightarrow Int\\ &succ :: Int^{S} \rightarrow Int\end{aligned}$$

In fact the x parameter to the *app* function is *always* evaluated if the f function is strict. In other words: *app* is polymorphic in the strictness of x, and the choice for S or L is same on the choice for S or L in the first parameter of f.

We call this polymorphism in annotations: *polyvariance*, as to not confuse it with normal type polymorphism. If we have a polyvariant strictness analysis, the following annotated type is inferred for *app*:

$$app :: (a^{\beta} \rightarrow b)^{S} \rightarrow a^{\beta} \rightarrow b$$

As with normal type polymorphism, all universal quantifications are implicit. We could write them out explicitly:

$$app :: \forall \beta.\ \forall a\ b.\ (a^{\beta} \rightarrow b)^{S} \rightarrow a^{\beta} \rightarrow b$$

In a language like TyCore, we turn all quantifications into an extra formal parameter, this would give is the following type:

$$app :: (\beta :: \varphi) \rightarrow (a :: \star) \rightarrow (b :: \star) \rightarrow (a^{\beta} \rightarrow b)^{S} \rightarrow a^{\beta} \rightarrow b$$

Note that we use φ as the type of a strictness property like L, S or a strictness property variable, like β.

At the call sites, we are now required to pass in three extra arguments; a strictness annotation and two types. Our two examples are now updated to the following:

$$\begin{aligned}&\mathit{app}\ L\ \mathit{Int}\ \mathit{Int}\ \mathit{foo}\ \ \ \bot\\&\mathit{app}\ S\ \mathit{Int}\ \mathit{Int}\ \mathit{succ}\ \bot\end{aligned}$$

5.3 A Dependent Type System?

In the previous example, we see that the strictness of the last argument to *app* depends on the value of the first argument. It could be argued that this is also the case in the *ite* function, lets look at a potential TyCore-like definition of that function:

$$\begin{aligned}&\mathit{ite} :: (a :: \star) \to (b :: \mathit{Bool}) \to (\_ :: a^{(\delta\ b)}) \to (\_ :: a^{(\delta\ (\neg\ b))} \to a\\&\mathit{ite}\ \_\ \mathit{True}\ \ x\ \_ = x\\&\mathit{ite}\ \_\ \mathit{False}\ \_\ y = y\end{aligned}$$

$$\begin{aligned}&\delta :: \mathit{Bool} \to \varphi\\&\delta\ \mathit{True}\ \ = S\\&\delta\ \mathit{False} = L\end{aligned}$$

Here, we have an "annotation level" function δ, that takes a boolean and computes a strictness property. Now, the strictness of the second and third arguments depend on the value of the first argument b.

However, if we think of strictness property as part of the type of a function (as TyCore does), we have a dependent type system. Currently we don't want to implement a fully dependent type system in TyCore, as this is a whole different area of research. So we don't want to allow this dependent type for *ite*.

How then do we allow for the polyvariant type of the *app* function? Isn't that also dependently typed?

5.3.1 Annotated type system

The answer lies in the definition of a "dependent type": A dependent type is a type that depends on a particular value. To resolve the issue of the dependent type, we simply state that β in the definition of *app* is not a value, but something else.

From now on, we refer to "things" like S, L or β as *annotations* instead of values. This way, the type of the last argument of *app* isn't dependent on a value, but on an annotation.

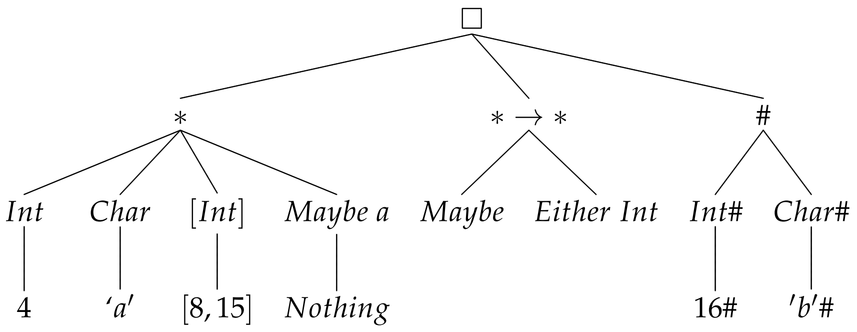
Granted, we now have moved the problem from a dependent type system to an annotated type system. But the scope of the problem is smaller, i.e. there are far fewer possible annotations then there are values, and in particular, we don't allow functions from values to annotations as our previous function δ did.

5.3.2 Kind system

To see how annotations fit in with things like values and types, we now look at Haskell's, or more accurately GHC's, value/type/kind system.

GHC kind system

We'll start of with an example that contains, different values, types and kinds:



- The items on the bottom level, are values. Conceptually, these are the things that exist at runtime, functions consume and produce values.
- One level above are the types of values. These separate the different values from each other, e.g. a function that takes an *Int* cannot be given a *Char*.
- Again one level up are the kinds, these are types of the types one level below.

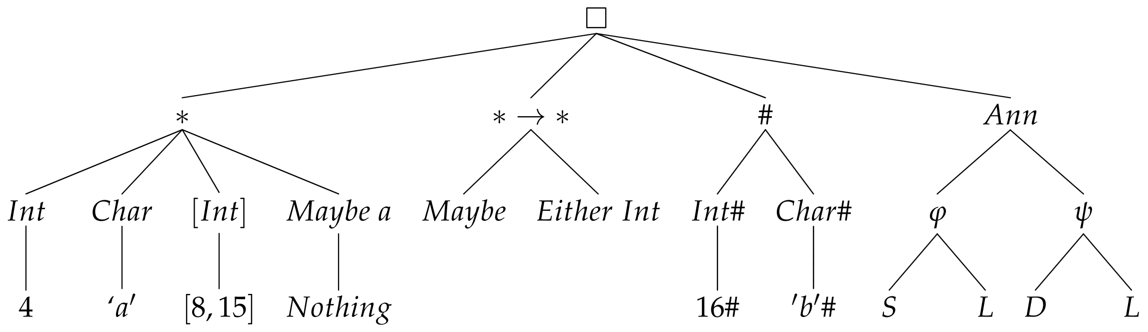
  Some types only exist at the type level and don't have any values, like the type constructor *Maybe* (which has kind $\star \rightarrow \star$).

Others have values, but they are completely separate from the "normal" values. For example, an unboxed int 3# has type *Int#*, which is of kind #. These unboxed types are in a separate kind from the usual boxed types. Therefor, the well known polymorphic identity function ($id :: \forall\, a :: \star.\ a \rightarrow a$) only works on values of a type that is of kind $\star$ and will not accept an unboxed character `'c'#`.

- Finally, at the top level of the tree, there is the level of super-kinds. Here, there is just a single super-kind: □, all kinds are of that super-kind.

Annotation kind system

Given the above diagram, we now extend it with a new kind *Ann*:



This diagram shows us how we can conceptually think of annotations as yet another kind of values in the system. All annotations are completely separated from "normal" values (as well as unboxed values), in the *Ann* kind.

At the type level, below the *Ann* kind, we see examples of two possible annotations; strictness analysis (φ) and an imaginary dead code analysis (ψ).

At the value level we see the familiar S and L values for strictness analysis. For the dead code analysis example, we see a dead annotation D and a liveness annotation L.

We can now see that, although annotations are conceptually on the level of values, they are separate from other values and require special attention in a compiler back-end. In the same way that an unboxed value like '3#' are separate from the boxed value '3' in GHC-Haskell. In the next section we will look at how we can translate annotation-values to a more low-level executable format.

5.4 Compiling Strictness

We now know the kinds of annotations that a polyvariant strictness analysis will infer. The types of function arguments are annotated with S for strict and L for non-strict. Also, annotation variables are used in polyvariant functions.

Given these annotations, we wish to optimize the code generated by the compiler to something that's more efficient. The principle of the optimization is simple: arguments at strict positions will be computed before entering the function body. Thus the result of evaluating the argument is passed to the function instead of a suspended computation.

This optimization is safe, because if the argument diverges ($\perp$), the function would also have. The optimized code is also faster, because no thunks have to be created and subsequently inspected and forced.

In this section we take an annotated TyCore-like language as our source language, and we translate this to GRIN as our low level language. GRIN is a good language to express optimized code because it makes evaluation order explicit. [1]

5.4.1 First-Order functions optimized

To see the different calling conventions in action, we first look at the *succ* example we saw before:

$$\begin{aligned}&\mathit{succ} :: \mathit{Int}^{S} \rightarrow \mathit{Int}\\&\mathit{succ}\ x = x + 1\end{aligned}$$

We also define two use sites of the function:

$\mathit{s1} = \mathit{succ}\ (1+2)$ -- should compute 4
$\mathit{s2} = \mathit{succ}\ (\mathit{fail}\ 3)$ -- should diverge (preferably, as early as possible)

$$\begin{aligned}&\mathit{fail} :: \mathit{Int}^{L} \rightarrow \mathit{Int}\\&\mathit{fail}\ \_ = \perp\end{aligned}$$

Lazy *succ* with no optimization

To see how the strictness optimization helps, we first look at the unoptimized version of the *succ* function. Below is a pretty printed snippet from a GRIN module produced by UHC:

[1] Technically, TyCore also has an explicit evaluation order, but translating to another language is a bit clearer, so we use GRIN here.

```
module LazySucc
  -- Globals
s1     ← store (Ffun_s1)
thunk1 ← store (Ffun_thunk1)
s2     ← store (Ffun_s2)
thunk2 ← store (Ffun_thunk2)
  -- Functions
succ x
  = store (CInt 1) ;  λonePtr →
    call plus x onePtr
fun_s1
  = call succ thunk1
fun_thunk1
  = store (CInt 1) ;  λonePtr →
    store (CInt 2) ;  λtwoPtr →
    call plus onePtr twoPtr
fun_s2
  = call succ thunk2
fun_thunk2
  = store (CInt 3) ;  λthreePtr →
    call fail threePtr
```

We will step through this snippet point-by-point:

- At the top of the module we see heap allocated globals.

  In our examples, *s1* and *s2* are in Constant Applicative Form (CAF), in GRIN this is implemented as a function with no arguments. Therefor, *s1* and *s2* are pointers to heap allocated tags pointing to *fun_s1* and *fun_s2*, respectively. The arguments to *s1* and *s2* (*thunk1* and *thunk2*) represent closures, and are also heap allocated.

- The *succ* function is relatively straightforward. It gets one argument *x*, constructs the integer 1, and calls the primitive *plus* function (not depicted here).

  Note that *succ* doesn't force its argument here, it is simply forwarded to *plus*, which will force the argument.

- The *s1* and *s2* CAFs are also simple, all they do is call *succ* with a delayed computation as argument.

- The delayed argument for *s1* is *fun_thunk1*, this function constructs the integers 1 and 2 and calls the primitive *plus* function.

- Similarly, the delayed argument *fun_thunk2* constructs the integer 3 and then calls the *fail* function (also not depicted).

Strict *succ* with optimization

We will now see how applying the strictness optimization changes the program:

```
module StrictSucc
  -- Globals
s1 ← store (Ffun_s1)
s2 ← store (Ffun_s2)
  -- Functions
succ x
   = store (CInt 1)            ; λonePtr →
     call plus x onePtr
fun_s1
   = store (CInt 1)            ; λonePtr →
     store (CInt 2)            ; λtwoPtr →
     call plus onePtr twoPtr ; λthreePtr →
     call succ threePtr
fun_s2
   = store (CInt 3)            ; λthreePtr →
     call fail threePtr        ; λres →
     call succ res
```

This program is already a lot shorter than the previous program, it no longer has the closure functions or the heap allocated pointers to the closures.

The bodies of the previous *fun_thunk1* and *fun_thunk2* functions are now inlined into *s1* and *s2*.

fun_s1 now constructs the ints 1 and 2 and calls *plus* to compute 3, this result is then passed as the argument to *succ*.

Similarly for *fun_s2*, this function constructs the int 3 and calls the *fail* function. *fail* will produce an error, and therefor the rest of the function won't be executed.

5.5 Optimizing Higher-Order Functions

We've seen how we can optimize a normal order function by evaluating a strict argument before entering the function body. As discussed before, this is more complicated when dealing with higher order functions.

We will look at some possibilities for optimizing higher order functions by means of an example:

$$\begin{array}{l} \textit{succAndTwice} :: (\textit{Int}^{\beta} \to \textit{Int}) \to \textit{Int}^{\beta} \to \textit{Int} \\ \textit{succAndTwice}\, f\ x = f\ (f\ (x+1)) \end{array}$$

The *succAndTwice* function has two parameters, a function f and an *Int* x. Whether x is strict or non-strict depends on the strictness of the function f.

Before looking at some uses of the function, we first introduce an explicit parameter for the strictness annotation β, as described in section 5.2.

$$\begin{array}{l} \textit{succAndTwice} :: (\beta :: \varphi) \to (\textit{Int}^{\beta} \to \textit{Int}) \to \textit{Int}^{\beta} \to \textit{Int} \\ \textit{succAndTwice}\ \_ f\ x = f\ (f\ (x+1)) \end{array}$$

We look a two use cases of this function that pass their strictness annotations explicitly:

$$\begin{array}{l} \textit{succ} :: \textit{Int}^{S} \to \textit{Int} \\ \textit{succ}\ x = x + 1 \\ \textit{foo} :: \textit{Int}^{L} \to \textit{Int} \\ \textit{foo} = \textit{ite}\ \textit{True}\ 1 \\ \textit{t1} = \textit{succAndTwice}\ L\ \textit{foo}\quad \{4+3\} \quad \text{-- Should return 1} \\ \textit{t2} = \textit{succAndTwice}\ S\ \textit{succ}\ (4+3) \quad \text{-- Should return 8} \end{array}$$

Note that we use braces ($\{\textit{expr}\}$) to represent a suspended computation.

5.5.1 Compiling explicit annotations

Now that we've introduced explicit annotations that are passed as arguments to functions, they have to be compiled somehow.

We explore two options:

Option 1: First class calling conventions

As per the title of Max Bolingbroke's paper "Types are calling conventions". Since the strictness annotations we've been working with are part of the types of the functions, they are also part of the calling conventions of the function. We've seen this in the *succ* example in the previous section. The lazy *succ* gets a pointer to a delayed computation, where as the strict *succ* gets the actual integer value. These are indeed different calling conventions.

Now that we have strictness annotations as arguments to functions, we would also need "first class calling conventions". That is, we would need to parameterize a function over its calling convention. Lets imagine two variants of the *succAndTwice* function, a lazy *succAndTwice′L* and a strict *succAndTwice′S*:

```
succAndTwice′L f x
  = store (CInt 1)                  ; λonePtr →
    store (Fplus x oncePtr)         ; λplusDelayedRes →
    store (AApply f plusDelayedRes) ; λapplyDelayedRes →
    apply f applyDelayedRes
succAndTwice′S f x
  = store (CInt 1)    ; λonePtr →
    call plus x onePtr; λplusRes →
    apply f plusRes   ; λfRes →
    apply f fRes
```

These two functions are obviously very different, the lazy function creates a bunch of delayed computations, and the strict version does each computation immediately. Now, lets look at the GRIN representation of the two uses of the higher order function:

```
fun_t1
  = call succAndTwice′L foo thunk1
fun_thunk1
  = store (CInt 4)             ; λfourPtr →
    store (CInt 3)             ; λthreePtr →
    call plus fourPtr threePtr
fun_t2
  = store (CInt 4)             ; λfourPtr →
    store (CInt 3)             ; λthreePtr →
    call plus fourPtr threePtr ; λsevenPtr →
    call succAndTwice′S succ sevenPtr
```

The call sites to the higher order *succAndTwice* are also different, the lazy function is called with a thunk, where as the strict function is called with a fully evaluated integer.

As we've seen the lazy and strict versions of higher order functions are radically different in their calling conventions as well as their implementations. While it may be possible to come up some way of doing first class calling conventions in machine code, that is certainly not expressible in the current GRIN.

If we want to add higher order calling conventions, we would have to change the mapping from TyCore to GRIN, as well as the GRIN language itself, as well as the mapping from GRIN to what ever is below GRIN, all the way down to the machine code. This is because no language (that the author is aware of) has first class calling conventions, so it would have to be build all the way up the stack.

Option 2: Partial evaluation

Instead of implementing first class calling conventions in a whole stack of languages, we will build an annotation partial evaluator. At the definition site of higher order functions, we specialize to remove annotations. At the use site of higher order functions, we run a partial evaluator to evaluate all uses of explicit annotations away.

All this can be done at the level of TyCore. For example, here are two specialized versions of *succAndTwice*:

$$\begin{aligned}
&\mathit{succAndTwice} : \langle\beta:\varphi\rangle \to \langle \mathit{Int}^{\beta} \to \mathit{Int}\rangle \to \mathit{Int}^{\beta} \to \mathit{Int}\\
&\quad = \lambda\langle\beta:\varphi\rangle\ \langle f : \mathit{Int}^{\beta} \to \mathit{Int}\rangle\ \langle x : \mathit{Int}^{\beta}\rangle \to\\
&\qquad \textbf{case}\ \beta\ \textbf{of}\\
&\qquad\quad L \to \mathit{succAndTwice}'L\ \langle f\rangle\ \{|x|\}\\
&\qquad\quad S \to \mathit{succAndTwice}'S\ \langle f\rangle\ \langle x\rangle\\
&\mathit{succAndTwice}'L : \langle \mathit{Int}^{L} \to \mathit{Int}\rangle \to \mathit{Int}^{L} \to \mathit{Int}\\
&\quad = \lambda\langle f : \mathit{Int}^{L}\rangle\ \langle x : \mathit{Int}^{L}\rangle \to f\ \{f\ \{x+1\}\}\\
&\mathit{succAndTwice}'S : \langle \mathit{Int}^{S} \to \mathit{Int}\rangle \to \mathit{Int}^{S} \to \mathit{Int}\\
&\quad = \lambda\langle f : \mathit{Int}^{S}\rangle\ \langle x : \mathit{Int}^{S}\rangle \to f\ \langle f\ \langle x+1\rangle\rangle
\end{aligned}$$

These to use sites with explicit annotations:

$$\begin{aligned}
t1 &= \mathit{succAndTwice}\ \langle L\rangle\ \langle \mathit{foo}\rangle\quad \{4+3\}\\
t2 &= \mathit{succAndTwice}\ \langle S\rangle\ \langle \mathit{succ}\rangle\ \langle 4+3\rangle
\end{aligned}$$

Are partially evaluated to:

$$\begin{aligned}
t1 &= \mathit{succAndTwice}'L\ \langle \mathit{foo}\rangle\quad \{4+3\}\\
t2 &= \mathit{succAndTwice}'S\ \langle \mathit{succ}\rangle\ \langle 4+3\rangle
\end{aligned}$$

Now that there are no more explicit annotations, we can use the existing TyCore to GRIN translation to compile this program.

Chapter 6

Implementation of TyCore Transformations

This chapter describes the implementation of five TyCore transformations implemented in UHC. Note that this chapter is intended as documentation for UHC developers. We will dive into details like filenames and AG attributes.

All files referred to in this chapter are located relative to `%UHC_HOME%/src/ehc`.

6.1 Language extension

Before talking about TyCore, we first look at a Haskell language extension. Since we haven't implemented a strictness inferencer, we need another way to instruct the strictness optimizer.

We've extended the Haskell type language with a new syntax for type based annotations. This is a subset of the grammar for types:

τ	$::=$	$\tau \rightarrow \tau$	Function type
	$\mid$	$\forall a.\tau$	Universal quantification
	$\mid$	a	Type variable
	$\mid$	**Con**	Concrete type
	$\mid$	$@a\ \ \tau$	Annotated type
	$\mid$	$@a\!:\!v\ \ \tau$	Type annotated with annotation variable

The last two lines have been implemented in the Haskell parser in `/HS-/Parser.chs` in the definition of `pTypeBase`. Annotations are parsed into a generic form, i.e. it doesn't matter what the annotation is, the Haskell parser will store it in the AST.

Since these changes to the parser affect the normal Haskell syntax, they are isolated in a separate UHC aspect called *tauphi*. Only when the

`--enable-tauphi` switch is provided to the `./configure` script, will this extension to the parser be enabled.

The `TypeAnnotation` data type in `/HS/AbsSyn.cag` has been extended with two constructors; `AnnotationName`, for the *@strict Int* form, and `AnnotationVar` for the *@strictness* : β *Char* form.

In the transformation to EH (`/HS/ToEH.cag`), the generic annotation representation is restricted to a more specific data type. The strings "`strict`", "`nonStrict`", and "`strictness`" are matched. They are transformed into *Strictness* (*Strict*), *Strictness* (*NonStrict*), *Strictness* (*StrictnessVar var*), respectively.

The *Strictness* constructor is an addition to the *TyExprAnn* data type in `/EH/AbsSyn.cag`. The *Strict* and *NonStrict* constructors are part of the *Strictness* data type in `/Base/Strictness.chs`. These constructors, as well as all other code described in this chapter are in variant 8, aspect *codegen* of UHC.

Once the annotations are in EH, they are transformed to Ty, in the `/EH/InferTyExpr.cag` transformation. From Ty on, the annotations are again transformed to TyCore in the `/Ty/ToTyCore.cag` transformation. Both Ty and TyCore have a new *Strictness* constructor in the *TyAnn* (`/Ty/AbsSynCore.cag`) and *ExprAnn* (`/TyCore/AbsSyn.cag`) data types.

The type checkers for Ty (`/Ty/FitsIn.chs`) and TyCore (`/TyCore-/Check.chs`) both simply ignore the annotations, and check the inner types.

6.2 Common

The file `/TyCore/Trf/Common.chs` contains some data types, functions, and type synonyms used across the different transformations.

6.2.1 BindType data type

Because TyCore uses a unified *Expr* data type, it can be hard to see where in an attribute grammar a semantic function is called. Several transformations use an inherited attribute *bindType* : *BindType* to get information about the location in the TyCore AST.

The *BindType* data type is defined as such:

$$\textbf{data}\ \mathit{BindType} = \mathit{NameTypeBind} \mid \mathit{BodyBind} \mid \mathit{NoBind}$$

A common TyCore AST has the form:

$$\begin{array}{l} \mathit{ValBind\_Val}\ (\mathit{Expr\_Seq}\ [...]) \\ \qquad\qquad\quad (\mathit{Expr\_Seq}\ [...]) \end{array}$$

The *bindType* attribute is used in *Expr_Seq*, to see if the current sequence of expressions is a name binding, or body definition. Deeper inside the structure, the inherited attribute is updated to *NoBind*, to indicate that nested *Expr_Seq*s aren't in bindings.

6.2.2 WorkWrap data type

Several transformations use a generated attribute *workWrap* : *WorkWrap*, this indicates the phase of the worker/wrapper transformation described in chapter 4. The *WorkWrap* data type is defined as such:

```
data WorkWrap
   = Introduced        -- A worker and wrapper are introduced
   | UpdatedWorker     -- A worker is updated
   | UpdatedWrapper    -- A wrapper is updated
   | Ignored           -- Nothing is done
```

These transformations also contain the generated attributes: *worker* : *Expr* and *wrapper* : *Expr*. The attributes are used to construct either new workers or wrappers, or to updated existing workers and wrappers.

At the top level of a binding, that is, when the *bindType* is set to *NameTypeBind* or *BodyBind*, functions types and definitions can be updated. When the *workWrap* attribute is set to *Introduced* the function is completely replaced by the values of the *worker* and *wrapper* attributes. A function that is an *UpdatedWorker* or *UpdatedWrapper* is replaced by either the *worker* or *wrapper*.

6.2.3 Seq type

Bindings in TyCore have the following form:

$$\begin{array}{l} \langle name_1 : type_1, name_2 : type_2\rangle \\ \quad = \langle expr_1, expr_2\rangle \end{array}$$

In the attribute grammars for the transformations, we often need to "bring" information from one side of the binding to the other. For example; Inside the semantic functions for $expr_1$, we may want to refer to $name_1$. Or in the functions for $type_2$, we want to refer to the number of **let** expressions in $expr_2$.

To accomplish this, attributes like *workWrap*, have sequence counterparts: *workWrapSeq* : *Seq WorkWrap*. *Seq a* is just a type synonym for $[a]$, but the different name helps to differentiate it from normal list type attributes.

6.3 Transformations

6.3.1 IntroduceExplicitLazyness

This transformation is implemented in the file `/TyCore/Trf/Introduce-ExplicitLaziness.cag`.

The standard translation from EH to TyCore will use the Strict Core convention for encoding thunks. At all places where laziness is introduced (all arguments to functions), a nullary function is created. Forcing a thunk is done by applying the nullary function to zero arguments:

Strict Core style		TyCore style	
$\langle\rangle \rightarrow \langle\tau_1, \ldots, \tau_n\rangle$	$\triangleq$	$\{\tau_1, \ldots, \tau_n\}$	Lazy type
$\lambda\langle\rangle \rightarrow e$	$\triangleq$	$\{e\}$	Delayed expression
$e\ \langle\rangle$	$\triangleq$	$\lvert e\rvert$	Forced expression

Figure 6.1: Strict Core and TyCore explicit laziness differences

The following example demonstrates the transformation, originally the TyCore program has this form:

$$\begin{aligned}&id : \langle\langle\rangle \rightarrow Int\rangle \rightarrow \langle Int\rangle\\&\quad = \lambda\langle x : \langle\rangle \rightarrow Int\rangle \rightarrow \langle x\ \langle\rangle\rangle\end{aligned}$$

To make it easier to later on write transformations that deal with laziness, we introduce explicit, *Lazy*, *Delay* and *Force* constructs. The transformation changes the previous example to:

$$\begin{aligned}&id : \langle\{Int\}\rangle \rightarrow \langle Int\rangle\\&\quad = \lambda\langle x : \{Int\}\rangle \rightarrow \langle\lvert x\rvert\rangle\end{aligned}$$

Note that the *Lazy* construct is used to encode lazy types. The *Delay* and *Force* constructs are used at the value level.

6.3.2 EliminateExplicitLazyness

This transformation is implemented in the file `/TyCore/Trf/Eliminate-ExplicitLaziness.cag`.

Since we haven't updated the TyCore type checker to deal with the *Lazy*, *Delay* and *Force* constructs, we remove these constructors from the final output. We apply the reverse transformation to the one described in the previous section.

In the TyCore transformation driver (`/TyCore/Trf.chs`), we apply *IntroduceExplicitLaziness* as the very first transformation, and *EliminateExplicitLaziness* as the very last. All other transformations happen in between, such that they can use the three constructs.

6.3.3 RemoveLazyFunctions

This two-part transformation is implemented in the files `/TyCore-/Trf/RemoveLazyFunctions1of2.cag` and `/TyCore/Trf/RemoveLazy-Functions2of2.cag`.

This, as well as several of the following transformations, are implemented in two parts. The first half of the transformation updates functions, the second half, updates their call sites.

To update call sites, the second half of the transformation uses the inherited attribute $tyEnv : Map\ HsName\ Ty$. This attribute is created in the file `/TyCore/Trf/ConstructTypeEnvironment.cag`. Because the transformations are split in two files, the *tyEnv* attribute will contain the updated types generated by the first half of the transformation.

The *RemoveLazyFunctions* transformation removes some unneeded laziness, created by the transformation from Essential Haskell to TyCore. All bindings in TyCore are generated as thunks to maintain Haskells concept of lazy binding:

$$\begin{aligned}&foo : \{Int\}\\&\quad = \{bar\ \langle 3\rangle\}\end{aligned}$$

This ensures that the call to *bar* is only evaluated when *foo* is used. However, this also generates thunks for functions, this is not needed, as functions are already in weak-head normal from:

$$\begin{aligned}&id : \{\langle\{Int\}\rangle \to \langle Int\rangle\}\\&\quad = \{\lambda\langle x : \{Int\}\rangle \to \langle|x|\rangle\}\end{aligned}$$

This unnessecery laziness is removed in *RemoveLazyFunctions1of2.cag*, by removing all thunks around lambda expressions. Note that this is a "definition based" transformation, not "type based".

In *RemoveLazyFunctions2of2.cag* the updated type for *id* is found in the *tyEnv* attribute and the call-sites are updated. Note that this transformation does not use the worker/wrapper method and will not work if TyCore were to be extended with modules.

6.3.4 DefinitionSiteArityRaising

This two-part transformation is implemented in the files `/TyCore/Trf-/DefinitionSiteArityRaising1of2.cag` and `/TyCore/Trf/Definition-SiteArityRaising2of2.cag`.

This transformation is described in section 4.1.1. An example of arity raising:

$$\begin{aligned}&\mathit{const} : \langle \mathit{Int}\rangle \to \langle \mathit{Char}\rangle \to \langle \mathit{Int}\rangle\\ &\quad = \lambda\langle x : \mathit{Int}\rangle\ \langle c : \mathit{Char}\rangle \to \langle x\rangle\end{aligned}$$

This example program is transformed to use the worker/wrapper method described in chapter 4 into:

$$\begin{aligned}&\mathit{const} : \langle \mathit{Int}\rangle \to \langle \mathit{Char}\rangle \to \langle \mathit{Int}\rangle\\ &\quad = \lambda\langle x : \mathit{Int}\rangle\ \langle c : \mathit{Char}\rangle \to \mathit{const}_{\mathit{worker}}\ \langle x, y\rangle\\ &\mathit{const}_{\mathit{worker}} : \langle \mathit{Int}, \mathit{Char}\rangle \to \langle \mathit{Int}\rangle\\ &\quad = \lambda\langle x : \mathit{Int}, c : \mathit{Char}\rangle \to \langle x\rangle\end{aligned}$$

The first part of the transformation creates or updates the worker. At this point the wrapper does not match the new type signature of the worker. In the second part, the wrapper is updated to the new type signature of the worker.

Note that even if a new wrapper is generated in the first part, it still doesn't have the correct call to the worker. So whether the worker is newly introduced or updated, the second part of the transformation still needs to take place.

6.3.5 OptimizeStrictness

This two-part transformation is implemented in the files `/TyCore-/Trf/OptimizeStrictness1of2.cag` and `/TyCore/Trf/Optimize-Strictness2of2.cag`.

This tranformation is described in section 4.1.2. An example of first order strictness optimization:

$$\begin{aligned}&\mathit{const} : \{@\mathit{strict}\ \mathit{Int}\} \to \{\mathit{Char}\} \to \langle \mathit{Int}\rangle\\ &\quad = \lambda\langle x : \{@\mathit{strict}\ \mathit{Int}\}\rangle\ \langle c : \{\mathit{Char}\}\rangle \to |x|\end{aligned}$$

This example program is transformed to use the worker/wrapper method described in chapter 4 into:

$$\begin{aligned}
&\mathit{const} : \{\mathit{Int}\} \to \{\mathit{Char}\} \to \langle \mathit{Int} \rangle \\
&\quad = \lambda \langle x : \{\mathit{Int}\} \rangle\ \langle c : \{\mathit{Char}\} \rangle \to \\
&\qquad \mathbf{let}\ x_0 = |x|\ \mathbf{in} \\
&\qquad \mathit{const}_{\mathit{worker}}\ \langle x_0 \rangle\ \langle y \rangle \\
&\mathit{const}_{\mathit{worker}} : \langle \mathit{Int} \rangle \to \{\mathit{Char}\} \to \langle \mathit{Int} \rangle \\
&\quad = \lambda \langle x : \mathit{Int} \rangle\ (c : \{\mathit{Char}\}) \to \langle x \rangle
\end{aligned}$$

The two-way split between the transformations is the same as for the *DefinitionSiteArityRaising* transformation. The first part creates or updates the worker, the second part updates the wrapper.

6.3.6 IntroduceWeirdConstructs

This transformation is implemented in the file `/TyCore/Trf/Introduce-WeirdConstructs.cag`.

Technically, this isn't a transformtion, but rather a set of manually created test cases. This file adds a set of TyCore expressions, to the module being compiled. These expressions are valid TyCore constructs, who are designed to test the previous transformations.

Since the TyCore code generated by Haskell is only a subset of all possible TyCore expressions, this transformation introduces some constructs that would not normally occur.

Examples of such expressions are:

$$\begin{aligned}
&\mathit{foo} : \langle \mathit{Int} \rangle \to \langle \mathit{Char}, \mathit{Char} \rangle \to \langle \mathit{Int} \rangle \to \langle \mathit{Int} \rangle \\
&\quad = \lambda \langle x : \mathit{Int} \rangle \to \\
&\qquad\quad \mathbf{let}\ y : \mathit{Int} = 3\ \mathbf{in} \\
&\qquad\quad \lambda \langle c : \mathit{Char}, d : \mathit{Char} \rangle\ \langle z : \mathit{Int} \rangle \to \langle x \rangle
\end{aligned}$$

This tests the *DefinitionSiteArityRaising* transformation, it should merge the $\langle \mathit{Char}, \mathit{Char} \rangle$ with the following $\langle \mathit{Int} \rangle$. In a normal Haskell program, a $\langle \mathit{Char}, \mathit{Char} \rangle$ wouldn't naturally occur.

$$\begin{aligned}
&\mathit{bar} : \langle \mathit{Int}, \{@\mathit{strict}\ \mathit{Char}, @\mathit{strict}\ \mathit{Char}\} \rangle \to \langle \mathit{Int} \rangle \\
&\quad = \lambda \langle x : \mathit{Int}, y : \{@\mathit{strict}\ \mathit{Char}, @\mathit{strict}\ \mathit{Char}\} \rangle \to \langle x \rangle
\end{aligned}$$

This tests the *OptimizeStrictness* transformation, it should completely remove all laziness to create the type: $\langle \mathit{Int}, \mathit{Char}, \mathit{Char} \rangle$. This means also means, creating two bindings for the one y binding that exists now.

Chapter 7

Conclusions

7.1 TyCore

In this thesis we've discussed the new TyCore intermediate language for the Utrecht Haskell Compiler. The language is particularly well suited for doing optimizing transformations for a typed, lazy, functional language. Two properties make TyCore a good intermediate language: Explicit laziness, multiple arguments and results.

A language that is lazy by default with some manual strictness (*seq*) is very useful for end users, but for an intermediate language, the reverse is better. Since the machine language to which every program ultimately compiles is strict, it makes sense to have the intermediate language also be strict. By making laziness explicit, we can reason over it: see where it is unnecessary and remove it.

The fact that TyCore functions take multiple simultaneous arguments and produce multiple results also closely matches the machine language. In the final assembly, arguments to subroutines are stored in machine registers and on stack locations. Having the intermediate language match this paradigm allows for transformations inside the language that optimize programs.

In short, TyCore is an interesting new language, that uniquely combines high-level features (i.e.: being a superset of λ-calculus) and supporting lower level features like multiple arguments and results.

7.2 Combining Optimizations

In compilers we often implement multiple optimizing transformations. Each of these transformations uses the output of the previous transformation as its input. When dealing with modularity through separate compilation, this pipeline of transformations can be difficult. Transformation authors would like to think of their transformation in isolation, but in reality they have to work together.

We've demonstrated how we can combine multiple optimizing transformations in such a way that none of the transformations has to be aware of the others. Instead of requiring all transformations to be familiar with the output of all the others, we only require them to be familiar with the worker/wrapper transformation. By using this shared worker/wrapper base knowledge, we can write transformations that can act like they operate in isolation. To our knowledge, this application of the worker/wrapper transformation has not been discussed before.

7.3 Polyvariant Strictness Optimization

We have discussed an extension to the TyCore language to enable strictness optimization for higher-order functions. By making strictness properties into first class values, we can reason over them. Since there is no way to actually implement strictness properties as values at runtime we need a way to remove them before code generation.

By putting "annotation values" in a different kind from normal values, we have found an easy way to distinguish between them. We use a partial evaluator at link time, to completely evaluate everything in the annotation kind away.

This work so far will be presented in the talk "Strictness Optimization for Higher-Order Functions in a Typed Intermediate Language" at the Implementation and Application of Functional Programming conference (IFL) in September 2010. We hope to explore this work in more detail in a future IFL paper.

7.4 Future work

The TyCore language already has shown itself useful for several optimizing transformations, we hope to see this extended to many more transformations in the future. We hope to fully implement the strictness optimization for higher-order functions in the Utrecht Haskell Compiler.

It is interesting to explore the concept of first class "strictness properties" further. What if we extend this to data types? What would a standard Haskell style list definition, with polyvariant strictness look like? Maybe something like: $\textbf{data}\ \mathit{List}\ (a : \star)\ (b : \varphi)\ (c : \varphi) = \mathit{Nil} \mid \mathit{Cons}\ a^{b}\ (\mathit{List}\ a\ b\ c)^{c}$

Depending on the instantiation, the compiler could generate different things, maybe a list of unboxed values, or an array of pointers, or even an array of unboxed values.

Furthermore it might be interesting the explore the applicability of the TyCore language outside the context of a Haskell compiler. The language might be general enough serve as a common base language between several high level functional language. In a similar way that .NET's Common Intermediate Language (CIL) is the lowest common denominator between several imperative, object oriented languages.

7.5 Conclusion

In this thesis we have explored the TyCore intermediate language. We've demonstrated its capabilities, and have shown how we can implement optimizing transformations on the language. We feel the language exists on a very interesting point in the spectrum between low and high level languages, and hope to see future use of the language in compiler development.

Acknowledgements

First of all, I want to thank Atze Dijkstra for supervising my work. I'm thankful for his continued help with both technical and non-technical problems.

I want to thank Doaitse Swierstra, Jeroen Fokker, Arie Middelkoop and all the others for the fun weekly UHC meetings. I've learned a lot during these meetings and I've gotten some valuable feedback on my own presentations.

A big 'thank you' to Stefan Holdermans, for interesting me in type based program analysis and for one day walking into the lab asking: "Have you thought about higher-order functions?".

Thanks to Matthijs van der Lit for checking my thesis for spelling and grammar mistakes. Also thanks to Jeroen Weijers and Jeroen Leeuwestein for their help on several technical problems, and for listening to my rants. Many thanks to Chris Eidhof, Sebastiaan Visser and Erik Hesselink for all the interesting discussions we've had about functional programming and Haskell. Special thanks to everyone in the ST lab for the always fun distractions, it might have slightly delayed my work, but I only have myself to blame for that.

Finally I would like to thank my family and friends for supporting me in what sometimes seemed like a never-ending process.

Bibliography

[1] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[2] Maximilian C. Bolingbroke and Simon L. Peyton Jones. Types are calling conventions. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 1–12, New York, NY, USA, 2009. ACM.

[3] Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *In Proc IFL 96, volume 1268 of LNCS*, pages 58–84. Springer-Verlag, 1996.

[4] Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Utrecht University, The Netherlands, 2005. ISBN 90-393-4070-6.

[5] Atze Dijkstra. Utrecht Haskell Compiler release announcement, 2009. `http://www.cs.uu.nl/wiki/UHC/Announce`.

[6] Andy Gill and Graham Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Prentice Hall, 2005.

[8] Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. A generic usage analysis with subeffect qualifiers. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 235–246, New York, NY, USA, 2007. ACM.

[9] Stefan Holdermans and Jurriaan Hage. Making "Stricterness" More Relevant. In *PEPM*, pages 121–130. ACM, 2010.

[10] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*, pages 39–50, New York, NY, USA, 2009. ACM.

[11] Simon Marlow. *Haskell 2010 Language Report*. 2010.

[12] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2004. ISBN 3-540-65410-0.

[13] Simon Peyton-Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[14] Simon Peyton-Jones and Erik Meijer. Henk: A Typed Intermediate Language. In *In Proc. First Int'l Workshop on Types in Compilation*, 1997.

[15] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

[16] Doaitse Swierstra et al. Utrecht University Attribute Grammar system, 2008. `http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem`.

[17] Andrew Tolmach and Tim Chevalier. An External Representation for the GHC Core Language, 2009.

[18] Edsko Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 201–218, Berlin, Heidelberg, 2008. Springer-Verlag.