A Constraint-based Bottom-up Strictness Analysis for Haskell

Master's Thesis

Lukas Spee

Institute of Information and Computing Sciences, Utrecht University

September 23, 2004

Acknowledgements

I would like to thank the following people for their contributions to this project:
Jurriaan Hage and Doaitse Swierstra for their supervision.
Bastiaan Heeren for all suggestions and help with the implementation.

Abstract

The Top project at Utrecht University has brought us a type inferencer which has customizability and meaningful error reporting as its most notable characteristics. At the heart of this inferencer lies a constrained-based bottom-up type inference algorithm. It is worth examining whether we can reuse some of the work on this type inferencer's design to implement other program analyses with similar properties. Such an analysis would collect the desired information as a type and effect system in the form of constraints on types. A first candidate for implementation is strictness analysis. However, existing algorithms for constraint-based strictness inference work in a top-down fashion. In this thesis we present a bottom-up variant of these algorithms that can handle a substantial subset of the Haskell98 language.

Chapter 1

Introduction

A well designed program analysis implementation returns useful feedback to the user if an inconsistency is detected, so that the user can easily track and correct the source of the error. An example is the type inferencer that is used by the Helium compiler developed at Utrecht University [6, 4]. This compiler has been designed to teach Haskell to novice functional programmers who may be unaware of the pitfalls in this language. It can hint the user on common mistakes. For example, if an error occurs because a function has the explicit type declaration `Int` → `Bool` → `Bool` and an inferred type `Bool` → `Int` → `Bool` the compiler can advise the user to switch the arguments of the function since the type error may have been caused by supplying the arguments to the function in the wrong order.

The inferencer combines information in the form of *constraints* from various locations in the abstract syntax tree of a Haskell program and solves these constraints at the top level. By combining constraints from different nodes of the abstract syntax tree we can obtain more information about an error than by a more local approach that tests each node of the abstract syntax tree individually for type correctness. This makes it, for example, possible to report the location where an error originates from, rather than the location where the error has been detected (which need not be the same).

Helium's type inferencer is the first and so far only program analysis that has been implemented as part of the Top project [13], which aims at developing a program analysis framework that is flexible and capable of producing meaningful feedback. In the case of the type inferencer this is achieved by, amongst others, using a constraint-based bottom-up type inference algorithm. In this thesis we present a *strictness analysis* based on a similar strictness inference algorithm. This algorithm is inspired by related work of Glynn *et al.* [3].

The rest of the thesis is structured as follows. In Chapter 2 the type inferencer of Top is discussed. Chapter 3 serves as a general introduction to strictness analysis. In Chapter 4 a top-down variant of the constraint-based strictness algorithm is discussed; in Chapter 5 the bottom-up variant is introduced. In Chapter 6 we shall see how the constraints generated by the algorithm of Chapter 5 can be solved. Chapter 7 discusses our implementation and Chapter 8 concludes and discusses future work.

The reader is assumed to have a basic knowledge of Haskell or a similar functional programming language. A complete overview of Haskell, including tutorials and definitions of the language and its standard libraries, can be found on the Haskell Home Page [11].

Chapter 2

An introduction to Top's type inferencer

Introduction

It is useful to first have a glance at Top's type inferencer [4, 5] since the implementation of the strictness inferencer is largely based on it. The Top framework consists of a number of modules, written in Haskell, that offer a.o. the following facilities

- Representation of types and type schemes (as data types)
- Representation of constraints (as data types)
- Constraint solvers based on various strategies (greedy, non-greedy, type graphs)
- Unifications and substitutions on types, constraints, etc.
- States to record information during inference

As mentioned earlier on, customizability and meaningful error reporting are outstanding features of the type inferencer. In Section 2.2 we shall see how this has been accomplished. First the required syntax and notation is introduced in Section 2.1.

2.1 Preliminaries

We consider a representative subset of the Helium language, which itself is a subset of Haskell, in this chapter. This subset is given by the following syntax.

$$
\begin{array}{lcl}
expr & ::= & lit \\
& | & var \\
& | & expr\ expr^{+} \\
& | & \text{'}\lambda\text{'}\ pat^{+}\ \text{'}\rightarrow\text{'}\ expr \\
& | & \texttt{if}\ expr\ \texttt{then}\ expr\ \texttt{else}\ expr \\
& | & \texttt{let}\ ((decl\ \text{';'}\ )^{*} decl)^{?}\ \texttt{in}\ expr \\
pat & ::= & var \\
decl & ::= & ((fb\ \text{';'}\ )^{*} fb)^{?} \\
& | & var\ \text{'::'}\ typescheme \\
fb & ::= & var\ pat^{*}\ \text{'='}\ rhs \\
rhs & ::= & expr
\end{array}
$$

The non-terminals *lit* and *var* refer to literals and variables, respectively. Note that an expression in the declaration of a let expression can be explicitly typed with a type scheme. The syntax for type schemes is

$$
\begin{array}{lcl}
\sigma & ::= & \tau \\
& | & \forall \overline{\alpha}.\tau \\
\tau & ::= & \alpha \\
& | & T\ \tau_1 \ldots \tau_n \text{ where } arity(T) = n
\end{array}
$$

Here T represents a type constructor. Type constants like `Int` and `Bool` can be considered as nullary type constructors. Likewise, the function type $\rightarrow$ can be considered as a binary type constructor. Infix notation is used for this constructor so types like `Int` $\rightarrow$ `Int` $\rightarrow$ `Bool` can be written. Parentheses are omitted here; $\rightarrow$ is assumed to be right-associative. Since data types are not discussed in this chapter, nor in the chapters on strictness inference (Chapter 4 and 5) we shall see no other type constructors than the ones just mentioned. A type scheme binds a vector of polymorphic type variables $\overline{\alpha}$ to a universal quantifier.

A type variable that is not bound by a quantifier is a *free type variable*. Note that all type variables that occur in a type τ, written as $ftv(\tau)$, are free since quantifiers do not occur in types. The free type variables of a type scheme $\forall \overline{\alpha}.\tau$ are given by $ftv(\forall \overline{\alpha}.\tau) = ftv(\tau) \backslash \overline{\alpha}$ where $\backslash$ denotes set difference.

A type τ can be generalized to a type scheme $\forall \overline{\alpha}.\tau$ with respect to a set of monomorphic type variables M by binding all type variables in τ that do not occur in M. This can be formalized as follows.

$$generalize(M, \tau) =_{def} \forall \overline{\alpha}.\tau \text{ where } \overline{\alpha} = ftv(\tau) \backslash M$$

Likewise, a type scheme can be instantiated to a type by replacing the bound type variables with fresh ones. We shall write $[\alpha_1 := \tau_1, \ldots,\ \alpha_n := \tau_n]$ for the substitution that maps α_i to τ_i.

$$
\begin{array}{c}
instantiate(\forall \alpha_1 \ldots \alpha_n.\tau) =_{def} [\alpha_1 := \beta_1, \ldots, \alpha_n := \beta_n]\tau \\
\text{where } \beta_1, \ldots, \beta_n \text{ are fresh}
\end{array}
$$

In Section 2.2 constraints are generated that describe relations that must hold between types or type schemes.

$$\begin{array}{rcl} C & ::= & \tau \equiv \tau \\ & | & \tau \leq_M \tau \\ & | & \tau \preceq \sigma \end{array}$$

These forms of constraints are called *equality constraints*, *implicit instance constraints*, and *explicit instance constraints*, respectively.

An equality constraint $\tau_1 \equiv \tau_2$ expresses that the types τ_1 and τ_2 are in fact equivalent and should be unified at some point. A let expression introduces polymorphism and instance constraints are used to deal with this. If a type scheme σ has been explicitly given in a let definition the explicit instance constraint $\tau \preceq \sigma$ can be used to express that the type τ should be an instance of σ. If no explicit type scheme is given, the implicit instance constraint $\tau_1 \leq_M \tau_2$ can be used to express that τ_1 should be an instance of the type scheme obtained by generalizing over τ_2 with respect to M, which is approximately $generalize(M, \tau_2)$ (it is possible that additional variables turn out to be monomorphic before generalization has taken place). The lifting of constraints on types to sets of assumptions i.e., a variable together with a type variable, is defined by $(X \oplus Y) = \{\tau_1 \oplus \tau_2 \mid x : \tau_1 \in X, x : \tau_2 \in Y\}$ where $\oplus$ is either $\equiv$, $\leq_M$, or $\preceq$.

2.2 Phased type inference

The type inference process in Top consists of three distinct phases. By phasing the process in this fashion we can gain some freedom of implementation for the various parts of the analysis. For example, if we would like to experiment with a different algorithm to solve the constraints in the final phase, this would not affect the implementation of the previous phases. Only the implementation of the first phase is more or less fixed. However, the programmer can define specialized type rules in external compiler directives to overcome this [5]. This section is based on Heeren *et al.* [4]; the reader is referred to their paper for a more detailed discussion of Tops' type inferencer.

Phase 1: Generation of constraints

In the first phase constraints are generated in the nodes of the abstract syntax tree of a program, using a system of type inference rules. A typing rule has the form of a deduction rule, where the premises can be found above the horizontal line and the conclusion underneath. Typically, a typing rule involves judgements of the form

$$M, \mathcal{A}, C \vdash^{e}_{type} e : \tau$$

This should be read as: 'we can infer the type τ for expression e with the set of monomorphic variables M, the assumption set $\mathcal{A}$, and the constraint set C'.

The set M is passed in a top-down fashion. At every node in the abstract syntax tree, M contains the monomorphic type variables that were introduced by lambda abstractions (and possibly other monomorphic constructs) higher up in the tree. This information is necessary to correctly generalize a type to a type scheme when an implicit instance constraint is generated.

The sets $\mathcal{A}$ and C are passed upwards. The set C consists of the constraints that are generated during type inference. In the assumption set $\mathcal{A}$ assumptions on the types of free variables in an expression are stored. Every instance of a free variable in a scope is associated with a fresh type variable. For each instance an assumption of the form $x : \tau$ ('variable x is of type τ') is stored in $\mathcal{A}$ so all instances of the same variable in the same scope can be related at some point. The domain of an assumption set X is defined as $dom(X) = \{x \mid x : \tau \in X\}$ and the range of X is defined as $ran(X) = \{\tau \mid x : \tau \in X\}$.

In this section, we will illustrate the inference process by the following example program:

```
let id = \x -> x
in  \x -> if x == 1 then x else id (id x)
```

Note that this example contains a polymorphic let-defined function `id`.

Every node in the abstract syntax tree is labeled with a fresh type variable β and the appropriate constraints are generated. This is done in a bottom-up fashion. For instance, consider this type rule for literals.

$$(\text{Lit}_{type})\frac{literal : \tau}{M, \emptyset, \{\beta \equiv \tau\} \vdash^{e}_{type} literal : \beta}$$

Note that this rule generates an equality constraint which is passed upwards. Since a literal expression does not contain any variables no assumptions are generated at this node. For example, if this rule is applied to the Boolean literal `False` a fresh type variable τ_1 is introduced and the constraint $\tau_1 \equiv$ `Bool` is generated.

The rule for variables is as follows.

$$(\text{Var}_{type})\frac{}{M, \{x : \beta\}, \emptyset \vdash^{e}_{type} x : \beta}$$

When a variable x is encountered, a fresh type variable β is introduced to associate with this instance of x and the corresponding assumption is generated. No constraints are imposed, however. We hope to find more information about the exact type of x when we go up in the abstract syntax tree. Until then β remains unrestricted.

For application, it must be verified that the types of the actual arguments fit the argument types that are expected by the function that is applied to them. Furthermore, the type of the application must be the result type of the function. The inference rule for application thus is as follows.

$$(\text{App}_{type})\frac{\begin{array}{c} M, \mathcal{A}, C \vdash^{e}_{type} f : \tau \quad M, \mathcal{A}_i, C_i \vdash^{e}_{type} a_i : \tau_i \text{ for } 1 \leq i \leq n \\ C' = C \cup \bigcup_i C_i \cup \{\tau \equiv \tau_1 \to \ldots \to \tau_n \to \beta\} \end{array}}{M, \mathcal{A} \cup \bigcup_i \mathcal{A}_i, C' \vdash^{e}_{type} f\ a_1 \ldots a_n : \beta}$$

Given the rules for literals, variables and application we can now derive the inference tree in Figure 2.1 for the subexpression `x == 1`, and Figure 2.2 for the subexpression `id (id x)` from our example program. Note that we consider `x == 1` to be infix notation for the application `(==) x 1`.

$$\dfrac{\dfrac{}{M_1, \mathcal{A}_7, C_7 \vdash^e_{type} \texttt{(==)} : \tau_7}\ (\text{Var}) \quad \dfrac{}{M_1, \mathcal{A}_8, C_8 \vdash^e_{type} \texttt{x} : \tau_8}\ (\text{Var}) \quad \boxed{\Delta_0}}{M_1, \mathcal{A}_6, C_6 \vdash^e_{type} \texttt{x == 1} : \tau_6}\ (\text{App})$$

where Δ_0 is

$$\dfrac{\texttt{1} : \texttt{Int}}{M_1, \mathcal{A}_9, C_9 \vdash^e_{type} \texttt{1} : \tau_9}\ (\text{Lit})$$

$\mathcal{A}_6 = \mathcal{A}_7 \cup \mathcal{A}_8 \cup \mathcal{A}_9$
$\mathcal{A}_7 = \{\texttt{(==)} : \tau_7\}$
$\mathcal{A}_8 = \{\texttt{x} : \tau_8\}$
$\mathcal{A}_9 = \emptyset$
$C_6 = C_7 \cup C_8 \cup C_9 \cup \{\tau_7 \equiv \tau_8 \to \tau_9 \to \tau_6\}$
$C_7 = \emptyset$
$C_8 = \emptyset$
$C_9 = \{\tau_9 \equiv \texttt{Int}\}$

Figure 2.1: Inference tree for the subexpression `x == 1`

$$\dfrac{\dfrac{}{M_1, \mathcal{A}_{12}, C_{12} \vdash^e_{type} \texttt{id} : \tau_{12}}\ (\text{Var}) \quad \boxed{\Delta_1}}{M_1, \mathcal{A}_{11}, C_{11} \vdash^e_{type} \texttt{id (id x)} : \tau_{11}}\ (\text{App})$$

where Δ_1 is

$$\dfrac{\dfrac{}{M_1, \mathcal{A}_{14}, C_{14} \vdash^e_{type} \texttt{id} : \tau_{14}}\ (\text{Var}) \quad \dfrac{}{M_1, \mathcal{A}_{15}, C_{15} \vdash^e_{type} \texttt{x} : \tau_{15}}\ (\text{Var})}{M_1, \mathcal{A}_{13}, C_{13} \vdash^e_{type} \texttt{id x} : \tau_{13}}\ (\text{App})$$

$\mathcal{A}_{11} = \mathcal{A}_{12} \cup \mathcal{A}_{13}$
$\mathcal{A}_{12} = \{\texttt{id} : \tau_{12}\}$
$\mathcal{A}_{13} = \mathcal{A}_{14} \cup \mathcal{A}_{15}$
$\mathcal{A}_{14} = \{\texttt{id} : \tau_{14}\}$
$\mathcal{A}_{15} = \{\texttt{x} : \tau_{15}\}$
$C_{11} = C_{12} \cup C_{13} \cup \{\tau_{12} \equiv \tau_{13} \to \tau_{11}\}$
$C_{12} = \emptyset$
$C_{13} = C_{14} \cup C_{15} \cup \{\tau_{14} \equiv \tau_{15} \to \tau_{13}\}$
$C_{14} = \emptyset$
$C_{15} = \emptyset$

Figure 2.2: Inference tree for the subexpression `id (id x)`

$$\frac{\boxed{\Delta_2} \quad \dfrac{}{M_1, \mathcal{A}_{10}, C_{10} \vdash^e_{type} \texttt{x} : \tau_{10}}\ (\text{Var}) \quad \boxed{\Delta_3}}{M_1, \mathcal{A}_5, C_5 \vdash^e_{type} \texttt{if x == 1 then ...} : \tau_5}\ (\text{If})$$

where Δ_2 is the subtree from Figure 2.1 and Δ_3 is the subtree from Figure 2.2

$$\mathcal{A}_5 = \mathcal{A}_6 \cup \mathcal{A}_{10} \cup \mathcal{A}_{11}$$
$$C_5 = C_6 \cup C_{10} \cup C_{11} \cup \{\tau_6 \equiv \texttt{Bool}, \tau_{10} \equiv \tau_5, \tau_{11} \equiv \tau_5\}$$

Figure 2.3: Inference tree for the subexpression `if x == 1 then x else id (id x)`

The rule for conditional expressions is intuitive. The guard expression is always of type `Bool`. Furthermore, the types of both branches must be equal to each other and to the type of the whole conditional expression. This yields the following rule.

$$(\text{If}_{type})\frac{\begin{array}{c} M, \mathcal{A}_1, C_1 \vdash^e_{type} e_1 : \tau_1 \quad M, \mathcal{A}_2, C_2 \vdash^e_{type} e_2 : \tau_2 \\ M, \mathcal{A}_3, C_3 \vdash^e_{type} e_3 : \tau_3 \\ C = C_1 \cup C_2 \cup C_3 \cup \{\tau_1 \equiv \texttt{Bool}, \tau_2 \equiv \beta, \tau_3 \equiv \beta\} \end{array}}{M, \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3, C \vdash^e_{type} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \beta}$$

In Figure 2.3 the application of this rule to the conditional expression from our example program is depicted.

All variables that occur in the patterns bound by a lambda abstraction are monomorphic. Every occurence of a variable in one of the patterns is associated with a fresh type variable by an assumption in the assumption set $\mathcal{B}$. These type variables are added to the set of monomorphic type variables M that is passed to the body of the abstraction.

Instances of a variable in the body of the lambda abstraction are related to its occurences in the patterns by the set of equality constraints ($\mathcal{B} \equiv \mathcal{A}$). All assumptions on the variables bound by the lambda abtractions are removed from the assumption set $\mathcal{A}$ that is passed upwards. Judgements of the form

$$\mathcal{B}, C \vdash^p_{type} p : \tau$$

are used to infer the types of the patterns. This implies that there is an inference rule for patterns beside the rules for expressions. We omit this rule here, but we remark that it is similar to the rule for variables in expressions, the main difference being the omission of the set of monomorphic variables M.

$$(\text{Abs}_{type})\frac{\begin{array}{c} \mathcal{B}_i, C_i \vdash^p_{type} p_i : \tau_i \text{ for } 1 \leq i \leq n \quad \mathcal{B} = \bigcup_i \mathcal{B}_i \\ M \cup ran(\mathcal{B}), \mathcal{A}, C \vdash^e_{type} e : \tau \\ C' = \bigcup_i C_i \cup C \cup (\mathcal{B} \equiv \mathcal{A}) \cup \{\beta \equiv \tau_1 \to \ldots \to \tau_n \to \tau\} \end{array}}{M, \mathcal{A} \backslash dom(\mathcal{B}), C' \vdash^e_{type} (\lambda p_1 \ldots p_n \to e) : \beta}$$

Figure 2.4 shows how the types of all uses of the variable x in the body of the lambda abstraction `\x -> if x == 1 then x else id (id x)` are related to

$$\dfrac{\dfrac{}{\mathcal{B}_4, C_4 \vdash^{p}_{type} \texttt{x} : \tau_4}\ (\text{Pat}) \quad \boxed{\Delta_4}}{M_0, \mathcal{A}_3, C_3 \vdash^{e}_{type} \lambda\texttt{x -> if x == 1 ...} : \tau_3}\ (\text{Abs})$$

where Δ_4 is the subtree from Figure 2.3

$\mathcal{A}_3 = \mathcal{A}_5 \backslash \texttt{x} = \mathcal{A}_7 \cup \mathcal{A}_9 \cup \mathcal{A}_{12} \cup \mathcal{A}_{14}$
$\mathcal{B}_4 = \{\texttt{x} : \tau_4\}$
$C_3 = C_4 \cup C_5 \cup \{\tau_4 \equiv \tau_{10}, \tau_4 \equiv \tau_8, \tau_4 \equiv \tau_{15}, \tau_3 \equiv \tau_4 \to \tau_5\}$
$C_4 = \emptyset$
$M_1 = M_0 \cup \{\tau_4\}$

Figure 2.4: Inference tree for the subexpression `λx -> if x == 1 then x else id (id x)`

the type variable τ_4, which is additionally added to the set of monomorphic type variables that is passed to the body.

A let expression introduces polymorphism. If a let definition is explicitly typed, explicit instance constraints are generated for all uses of the defined expression in the body of the let expression. If the definition lacks an explicit type scheme, implicit instance constraints are generated instead. The presence of explicit type schemes determines in part which definitions belong to the same binding group. A let definition may only be used *monomorphically* in definitions within the same binding group, unless it has been explicitly typed with a polymorphic type scheme. Consider the following example.

```
let h :: a -> a
    f = ... g ... h ... j ...
    g = ... f ... g ...
    h = ... f ...
    j = ...
in ...
```

In Haskell type variables are implicitly bound by a universal quantifier; the type `a -> a` should be read as $\forall a.a \to a$. The exact details of the definitions and the body are not given, since these are not relevant here. A variable name occuring at the right-hand side of a definition denotes that the variable is used in that definition.

The relations between the definitions of `f`, `g`, `h`, and `j` are depicted in the directed graph of Figure 2.5. A path from node u to node v indicates that u is used in the definition of v. If there is a path from u to v and a path from v to u, both definitions belong to the same binding group (i.e., they are mutually recursive and can only be typed together). However, if an explicit type declaration is supplied for a definition x, this cuts off x from its original binding group B, because the information from the type declaration can be used to type the other definitions from B before typing x.

For our example, we can observe that `f` and `g` form a binding group together.

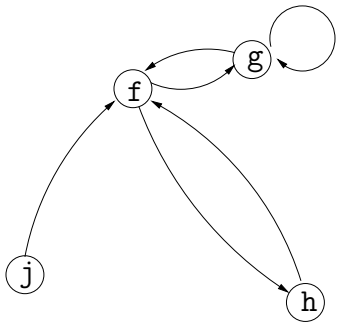

Figure 2.5: Graph representation of the relations between the let-defined functions `f`, `g`, `h` and `j`

Although `f` and `h` are also mutually recursive, `h` is not part of this binding group since it is explicitly typed. The typing order thus is: first `j`, then the binding group (`f`, `g`), and finally `h`.

We need to take the binding group information into account when we infer the type of a let expression. We therefore perform a binding group analysis on its let definitions to determine the binding groups and to obtain the appropriate assumptions and constraints given this information. The function *bga* in the following inference rule achieves this. Generalization of the types inferred for the definitions in the let expression is part of the binding group analysis.

$$(\text{Let}_{type})\frac{\begin{array}{c} M, \mathcal{B}_i, \mathcal{A}_i, C_i \vdash^{d}_{type} d_i \text{ for } 1 \leq i \leq n \quad M, \mathcal{A}, C \vdash^{e}_{type} e : \tau \\ (\mathcal{A}', C') = bga(M, explicits, \{(\emptyset, \mathcal{A}), (\mathcal{B}_1, \mathcal{A}_1), \ldots, (\mathcal{B}_n, \mathcal{A}_n)\}) \\ C'' = \bigcup_i C_i \cup C \cup C' \cup \{\beta \equiv \tau\} \end{array}}{M, \mathcal{A}', C'' \vdash^{e}_{type} \texttt{let } explicits; d_1; \ldots d_n \texttt{ in } e : \beta}$$

Just as is the case for patterns, there is a special inference rule for declarations, which is applied to the let definitions. Like the inference rules for patterns, the rule for declarations is omitted here. Figure 2.6 shows the application of $(\text{Let})_{type}$to our running example program. Details about the inference of the type of the let definition are left out. Note, however, the generated implicit instance constraints.

Phase 2: Ordering of constraints

After decorating the tree with constraints, the second phase consists of flattening the tree into an ordered list of constraints. The ordering of the constraints largely determines the location where an inconsistency is detected, provided that the solver respects this ordering. To specify the desired behaviour in ordering, several treewalk strategies are available to choose from. These include the well-known postorder treewalk $\mathcal{W}$ and its preorder variant $\mathcal{M}$.

Although the programmer is in principle free to experiment with various other treewalks to flatten the tree, there is one restriction on the ordering of the constraints that should be respected: before the constraints of the body of

$$\dfrac{\dfrac{\dots}{M_0, \mathcal{A}_2, C_2 \vdash^{d}_{type} \texttt{id = λx -> x} : \tau_2}\ \text{(Decl)} \quad \boxed{\Delta_5}}{M_0, \mathcal{A}_1, C_1, \vdash^{e}_{type} \texttt{let id = λx in ...} : \tau_1}\ \text{(Let)}$$

where Δ_5 is the subtree from Figure 2.4

$\mathcal{A}_1 = \dots$
$\mathcal{A}_2 = \dots$
$C_1 = C_2 \cup C_3 \cup \{\tau_{12} \leq_\emptyset \tau_2, \tau_{14} \leq_\emptyset \tau_2, \tau_1 \equiv \tau_2\}$
$C_2 = \dots$
$M_0 = \emptyset$

Figure 2.6: Inference tree for the expression `let id = λx -> x in λx -> if x == 1 then x else id (id x)`

a let expression are solved, first the constraints of outer and preceding scopes—including the let definitions—should have been solved. Only then do we know which variables are certainly monomorphic in the body of the let expression.

Phase 3: Solving of constraints

In the final phase the constraints are solved, possibly resulting in inconsistencies for which appropriate error messages are generated. Again, different implementations can be used for solving the constraints. Examples of these are greedy constraint solving and constraint solving with type graphs.

Conclusion

At the heart of Top's well-behaved type inferencer lies a constraint-based algorithm that can be formalized as a system of type inference rules that are applied to a program in a bottom-up fashion. In this chapter we have seen some of these rules for common constructs of Haskell, including polymorphic let expressions. The constraints that are generated by the inference algorithm are collected in an ordered list which is then checked for consistency.

There is a lot of freedom for the programmer in the implementation of the type inferencer. It is possible to add specialized type rules by means of external directives to the compiler. For the ordering and solving of the constraints several implementations are available, and one can experiment with selfmade implementations, as well.

Chapter 3

An introduction to strictness analysis

Introduction

In a lazy language (like Haskell, for example) the value of an argument to a function is normally evaluated when the body of the function is executed rather than at the call-site of the function, where the argument is passed to the function. Consider, for instance, the following function definition in Haskell.

```
f a b = if True then a else b
```

It is easy to see that the value of the argument `b` in fact will never be used in calculating the result of `f`. So, since Haskell is a lazy language, the value of `b` will never be evaluated[1]. This suggests that lazy evaluation usually is more efficient than *eager* evaluation (i.e., evaluation of the arguments at the call-site of the function). In practice, however, it turns out that this is generally not the case. Lazy evaluation takes up more space since calculations, rather than evaluated values, need to be stored. It is therefore worth examining which functions in a program can be eagerly evaluated.

Can't we just evaluate *all* functions in the program in an eager fashion? In general, the answer is no, since in doing so we might change the termination behaviour of the program. If we have another look at the function definition given above, we could imagine that the 'value' of `b` is, in fact, a non-terminating calculation. We can write this as $b = \bot$ or, in Haskell, `b = undefined`. An infinite data structure like `b = [0..]` is also a non-terminating calculation. Lazy evaluation of `f` will terminate (assuming that the evaluation of the value of `a` terminates) since the whole 'else'-branch of the conditional is skipped in the evaluation of `f`. Eager evaluation, on the other hand, will first evaluate the arguments of `f` before having a look at the body of this function. But since the evaluation of `b` will not terminate, the evaluation of `f` will result in a non-terminating calculation too.

[1] Throughout this paper we have taken the liberty to be a bit informal in speaking of 'the value of `b`'. Technically, this should be read as 'the value of the call argument corresponding to the formal argument `b`'.

A program transformation should be *safe*; it should not affect the external behaviour of the program. If eager evaluation of the argument `a` of a function `f` is safe, `f` is said to be *strict* in `a`. More formally, we can define `f` to be strict in its i^{th} argument iff

$$\mathtt{f}\ \mathtt{x}_1 \ldots \mathtt{x}_{i-1}\ \bot\ \mathtt{x}_{i+1} \ldots \mathtt{x}_n = \bot \qquad \forall \mathtt{x}_1, \ldots, \mathtt{x}_{i-1}, \mathtt{x}_{i+1}, \ldots, \mathtt{x}_n$$

Thus, the goal of strictness analysis is to determine whether a function is strict in any of its arguments. This is in general an intractable problem, considering that the domain of possible input values of the program may be infinite. There are, however, ways to come to a satisfactory approximation of the answer.

In this chapter we discuss strictness analysis based on abstract interpretation, as introduced by Mycroft [8]. Roughly speaking, abstract interpretation [2] abstracts over a value of a possibly infinite domain by representing the (concrete) value by an *abstract value* of a finite abstract domain. Which abstract domain to use, depends on the specific property we are interested in. An introduction to abstract interpretation can be found in the textbook by Nielson *et al.* [9]. For our purpose, we use the abstract domain $\mathbf{2} = \{\bot, \top\}$ with the partial ordering $\bot \leq \top$ to abstract over the domain of argument values. The abstract value $a^\#$ of the concrete value a can then be defined as follows.

$a^\# = \bot$ if a does not terminate
$a^\# = \top$ if it is unknown whether a terminates

Note that $\bot$ is used to represent the non-terminating calculation in both the concrete and the abstract domain[2]. The abstract domain $\mathbf{2}$ nicely fits the domain of Boolean values, mapping $\bot$ to the Boolean value *false* and $\top$ to *true*. The logical operations $\vee$ and $\wedge$ can be used to take the join and meet, respectively, of two values from $\mathbf{2}$.

An important restriction on the use of abstraction is that it should be safe. Safety in this case means that the outcome of an analysis on the abstract values should not give a wrong answer. The strictness information we get thus may be less informative than applying the analysis to concrete values, but it should not be inconsistent with it. We can formalize this as the *safety criterion*

$$[\![\cdot]\!]_\# \,.\, \mathtt{f} \ \leq\ \mathtt{f}^\# \,.\, [\![\cdot]\!]_\#$$

where the function $[\![\cdot]\!]_\#$ is the function that, when given a concrete value, produces the matching abstract value, and where $\mathtt{f}^\#$ is the abstract version of the concrete function `f`. If we can find such an $\mathtt{f}^\#$ that satisfies the safety criterion, we can lift the definition of strictness of `f` in its i^{th} argument, given earlier in this section, to the abstract domain.

$$\mathtt{f}^\#\ \top \ldots \top\ \bot\ \top \ldots \top = \bot$$

[2] In some literature on this subject the abstract domain $\mathbf{2}$ is written as $\{0, 1\}$. However, the values of that domain can be confused with the values 0 and 1 from the domain $\mathbb{Z}$. We therefore choose to follow Wadler [14] in using the proposed notation. Whether the abstract value $\bot$ or its concrete counterpart is meant, is usually clear from context.

By lifting the problem to the abstract world we have simplified it considerably: to check that a function is strict in a certain argument, we just fill in $\bot$ for this argument and $\top$ for all others in the body of the corresponding abstract function and verify that this is consistent with choosing the value $\bot$ for the function result. The question that remains is how to find a proper abstract function for a given concrete function. This is the main subject of the rest of this chapter. We start simple by looking at functions on flat, first-order domains (Section 3.1). Higher-order functions are considered next (Section 3.2), and after that we shall see an example of non-flat domains, namely lists (Section 3.3).

Since this chapter merely serves as an introduction to strictness analysis, we refrain from defining the underlying language in a formal way. Some common constructs in Haskell and many other functional programming languages are considered to give a feel of how strictness analysis by abstract interpretation works. The set of constructs is by no means complete; let expressions, for example, are omitted here.

3.1 Functions on flat, first-order domains

The functions that are discussed now have arguments of a simple type i.e., they are not themselves functions or instances of data types. Let's start with a very basic example.

```
incr x = x + 1
```

How to find $\texttt{incr}^\#$? Looking at the definition of `incr`, one can identify the variable `x`, the constant value `1` and the primitive binary operator `(+)` (arithmetic addition). We now need to find translations to abstract values for these classes of expressions. Evaluation of a constant will always terminate, so the translation for constants is straightforward.

$$[\![c]\!]_\# = \top$$

We have tacitly assumed here that $\bot$ is not a member of the class of constants. Here is the obvious translation for this somewhat special value.

$$[\![\bot]\!]_\# = \bot$$

Variables in the concrete domain are translated to variables in the abstract domain.

$$[\![x]\!]_\# = \texttt{x}^\#$$

The translation for binary arithmetic operators like addition and multiplication is also easy to deduce: both arguments of such an operator should terminate for the operation to terminate. For the `(+)`-operator the following abstraction is thus obtained.

$$[\![e_1 \texttt{ + } e_2]\!]_\# = [\![e_1]\!]_\# \wedge [\![e_2]\!]_\#$$

A similar translation can be made for the other primitive arithmetic operators. The abstract function $\texttt{incr}^\#$ can now be constructed.

$$\texttt{incr}^\# \ \texttt{x}^\# = [\![\texttt{x + 1}]\!]_\# = [\![\texttt{x}]\!]_\# \wedge [\![\texttt{1}]\!]_\# = \texttt{x}^\# \wedge \top = \texttt{x}^\#$$

To determine whether `incr` is strict, we fill in $\bot$ for `x` and immediately see that this yields $\texttt{incr}^\#\bot = \bot$. So, indeed, `incr` is strict in `x`.

Here is a more complicated function that uses `incr`.

```
incrNeg x = if x < 0 then incr x else x
```

To determine $\texttt{incrNeg}^\#$ we need to know how to abstract over conditionals and function applications. A conditional surely does not terminate if the condition is a non-terminating expression. Suppose that the condition *does* terminate, then it is only sure that evaluation of the conditional will not terminate if both branches are non-terminating. The translation scheme for conditionals becomes

$$[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]_\# = [\![e_1]\!]_\# \wedge ([\![e_2]\!]_\# \vee [\![e_3]\!]_\#)$$

The translation of function application is just as one would expect.

$$[\![e_1\ e_2]\!]_\# = [\![e_1]\!]_\#\ [\![e_2]\!]_\#$$

For $\texttt{incrNeg}^\#$ we find

$$\begin{array}{rcl}
\texttt{incrNeg}^\#\ \texttt{x}^\# & = & [\![\texttt{if x < 0 then incr x else x}]\!]_\# \\
& = & [\![\texttt{x < 0}]\!]_\# \wedge ([\![\texttt{incr x}]\!]_\# \vee [\![\texttt{x}]\!]_\# \\
& = & ([\![\texttt{x}]\!]_\# \wedge [\![\texttt{0}]\!]_\#) \wedge ([\![\texttt{incr}]\!]_\#\ [\![\texttt{x}]\!]_\# \vee [\![\texttt{x}]\!]_\#) \\
& = & (\texttt{x}^\# \wedge \top) \wedge (\texttt{x}^\# \vee \texttt{x}^\#) \\
& = & \texttt{x}^\#
\end{array}$$

Note that $\texttt{incr}^\#\ \texttt{x}^\#$ is substituted for $\texttt{x}^\#$, which follows from the definition of $\texttt{incr}^\#$.

Before the domain of higher-order functions is considered, recursive functions have to be dealt with first. Consider the function

```
rec x y = if x < 0 then x + y else rec (x - 1) y
```

If the translations described above are applied, this results in

$$\texttt{rec}^\#\ \texttt{x}^\#\ \texttt{y}^\# = \texttt{x}^\# \wedge ((\texttt{x}^\# \wedge \texttt{y}^\#) \vee [\![\texttt{rec (x - 1) y}]\!]_\#)$$

So an abstraction over `rec (x - 1) y` is needed to find the abstraction over `rec x y`. It seems we are stuck in a vicious circle. The solution is to perform a fixpoint computation. This computation is performed as follows: we start by approximating the recursive call of the function in the body by $\bot$. For the example above this yields

$$\texttt{rec}_1^\#\ \texttt{x}^\#\ \texttt{y}^\# = \texttt{x}^\# \wedge ((\texttt{x}^\# \wedge \texttt{y}^\#) \vee \bot) = \texttt{x}^\# \wedge \texttt{y}^\#$$

Here, $\texttt{rec}_n^\#$ indicates the value that was found for $\texttt{rec}^\#$ in the n^{th} iteration. We repeatedly substitute the recursive function call with the approximation that we have found in the previous iteration, until the approximation remains the same in two consecutive iterations. So, given that $\texttt{rec}_1^\#\ \texttt{x}^\#\ \texttt{y}^\# = \texttt{x}^\# \wedge \texttt{y}^\#$, we find in the second iteration that

$$\begin{array}{rcl}
\texttt{rec}_2^\#\ \texttt{x}^\#\ \texttt{y}^\# & = & \texttt{x}^\# \wedge ((\texttt{x}^\# \wedge \texttt{y}^\#) \vee (\texttt{rec}_1^\#\ \texttt{x}^\#\ \texttt{y}^\#)) \\
& = & \texttt{x}^\# \wedge ((\texttt{x}^\# \wedge \texttt{y}^\#) \vee (\texttt{x}^\# \wedge \texttt{y}^\#)) \\
& = & \texttt{x}^\# \wedge \texttt{y}^\#
\end{array}$$

which is the same approximation as the one we found in the previous iteration, so we are done and $\texttt{rec}^\# \ \texttt{x}^\# \ \texttt{y}^\# = \texttt{x}^\# \wedge \texttt{y}^\#$. Now we can determine whether `rec` is strict in its arguments.

$$\texttt{rec}^\# \ \bot \ \top = \bot \wedge \top = \bot$$

and

$$\texttt{rec}^\# \ \top \ \bot = \top \wedge \bot = \bot$$

Thus `rec` is strict in both `x` and `y`.

3.2 Higher-order functions

All functions that we discussed in Section 3.1 are first-order functions; they cannot have a function as an argument or return a function as a result. We shall now make an extension to the analysis so that we can cope with higher-order functions as well. This technique turns out to be relatively simple. For example, suppose we have the function `apply` defined as

```
apply f x = f x
```

where `x` is a flat first-order type. Abstracting over `apply` yields

$$\texttt{apply}^\# \ \texttt{f}^\# \ \texttt{x}^\# = \texttt{f}^\# \ \texttt{x}^\#$$

To determine whether `apply` is strict in its functional argument `f`, we need some abstraction over the function space $\mathbf{2} \to \mathbf{2}$ that represents the bottom value of this domain, like $\bot$ represents the bottom value of $\mathbf{2}$. The abstraction $\lambda\texttt{x}^\# \to \bot$ does the trick. Note that this abstract function always returns $\bot$, regardless of its argument. We get

$$\texttt{apply} \ (\lambda\texttt{x}^\# \to \bot) \ \top = (\lambda\texttt{x}^\# \to \bot) \ \top = \bot$$

So `apply` is strict in its argument `f`. To check whether `apply` is strict in its second argument, we can similarly abstract over `f` by $\lambda\texttt{x}^\# \to \top$ (the variant of $\top$ for the function space $\mathbf{2} \to \mathbf{2}$ which gives us

$$\texttt{apply} \ (\lambda\texttt{x}^\# \to \top) \ \bot = (\lambda\texttt{x}^\# \to \top) \ \bot = \top$$

This shows us that `apply` is not necessarily strict in `x`, which is the result that we would expect: we don't know if `f` really uses its argument `x` or just throws it away. If we would know the defintion of `f`, we could give a more precise approximation of its strictness (as is shown in Section 3.1).

The functional argument of `apply` in the example above has a flat, first-order argument and returns a flat, first-order result value. Thus the function space is, as mentioned before, $\mathbf{2} \to \mathbf{2}$. More generally, a function `f` in the domain $\mathbb{A} \to \mathbb{B}$, where $\mathbb{A}$ and $\mathbb{B}$ may have any height (i.e., these may be domains of n-ary functions, where $n \geq 0$), is undefined if its argument is undefined when

$$\texttt{f}^\# \ (\bot_{\mathbb{A}}) = \bot_{\mathbb{B}}$$

Here $\bot_{\mathbb{A}}$ and $\bot_{\mathbb{B}}$ are the bottom elements of the domains $\mathbb{A}$ and $\mathbb{B}$, respectively.

3.3 Functions on non-flat domains

Another useful extension is strictness analysis on non-flat domains. In this section we show how to abstract over the non-flat domain of lists. A more general method for data structures is not presented; opinions seem to differ on the complexity of such a generalization.

First, we consider lists of integers only. The abstract domain of such lists is actually a bit larger than what we are used to:

$$\begin{array}{lcl} as^{\#} & = & \top_{\in} \text{ if } as \text{ is a finite list, not containing the element } \bot \\ as^{\#} & = & \bot_{\in} \text{ if } as \text{ is a finite list, containing the element } \bot \\ as^{\#} & = & \infty \text{ if } as \text{ is (an approximation to) an infinite list} \\ as^{\#} & = & \bot \text{ if } as \text{ is a non-terminating calculation} \end{array}$$

Note that the value $\bot$ is once again overloaded: we also use it as the bottom value in the domain of lists of integers. The partial ordering of the domain is $\bot \leq \infty \leq \bot_{\in} \leq \top_{\in}$. The usual join operator $\sqcup$ and meet operator $\sqcap$ are used to obtain the least upper bound and the greatest lower bound, respectively, of two values from this domain. Furthermore, we assume that there are two (constructor) functions to build lists: `nil` represents the empty list (denoted in Haskell by `[]`), and `cons` is similar to the (`:`)-operator in Haskell. The corresponding abstractions over these functions are

$$\begin{array}{lcl} [\![\texttt{nil}]\!]_{\#} & = & \top \\ [\![\texttt{cons x xs}]\!]_{\#} & = & \texttt{cons}^{\#}\ \texttt{x}^{\#}\ \texttt{xs}^{\#} \end{array}$$

where the definition of $\texttt{cons}^{\#}\ \texttt{x}^{\#}\ \texttt{xs}^{\#}$ is given by the table

	$\texttt{x}^{\#} = \top$	$\texttt{x}^{\#} = \bot$
$\texttt{xs}^{\#} = \top_{\in}$	$\top_{\in}$	$\bot_{\in}$
$\texttt{xs}^{\#} = \bot_{\in}$	$\bot_{\in}$	$\bot_{\in}$
$\texttt{xs}^{\#} = \infty$	∞	∞
$\texttt{xs}^{\#} = \bot$	∞	∞

Most abstractions mentioned above are straightforward, but the notion of an approximation to an infinite list may not be clear at first sight. An example of such a list is `cons` 3 (`cons` 2 $\bot$). Although this list is in fact finite (it has a finite number of elements), the evaluation of the last element will not terminate. Even worse, since the list does not contain `nil`, the evaluator will not know that the list is finite. Thus the termination behaviour of this list resembles that of an infinite list. For the remainder of this paper we define an approximation to a infinite lists to be a list which has the value $\bot$ as its last element. A finite list, on the other hand, will always have `nil` as its last element. So, for example `cons` 3 (`cons` $\bot$ `nil`) is a finite list, although one of its members is $\bot$; its abstract value is $\bot_{\in}$.

Functions on lists can be defined using pattern matching. The general form of such a function is

```
h nil         = a
h (cons x xs) = f x xs
```

Here is, somewhat out of the blue, how we can define $\mathtt{h}^{\#}\ \mathtt{xs}^{\#}$.

$$\begin{array}{lcl}
\mathtt{h}^{\#}\ \top_{\in} & = & [\![a]\!]_{\#} \sqcup (\mathtt{f}^{\#}\ \top\ \top_{\in}) \\
\mathtt{h}^{\#}\ \bot_{\in} & = & (\mathtt{f}^{\#}\ \bot\ \top_{\in}) \sqcup (\mathtt{f}^{\#}\ \top\ \bot_{\in}) \\
\mathtt{h}^{\#}\ \infty & = & \mathtt{f}^{\#}\ \top\ \infty \\
\mathtt{h}^{\#}\ \bot & = & \bot
\end{array}$$

How did we obtain this definition? Let's first look at the case where the argument has value $\bot$. Pattern matching against this argument will not terminate, since its evaluation is a non-terminating calculation. Applying $\mathtt{h}^{\#}$ to $\bot$ thus yields $\bot$. For the case where the argument value is ∞, the argument always pattern matches with the `cons`-case of the definition of `h`. If we look at the table with the definition of `cons`, we see that the value ∞ can be obtained by applying `cons` to a flat argument with the abstract value $\top$ or $\bot$, and a non-flat argument with the abstract value $\bot$ or ∞. We now construct the abstract value of $\mathtt{h}^{\#}\ \bot$ by taking the least upper bound of these cases. Since $\bot \leq \top$ and $\bot \leq \infty$ we get

$$\begin{array}{lcl}
\mathtt{h}^{\#}\ \infty & = & (\mathtt{f}^{\#}\ \top\ \infty) \sqcup (\mathtt{f}^{\#}\ \bot\ \infty) \sqcup (\mathtt{f}^{\#}\ \top\ \bot) \sqcup (\mathtt{f}^{\#}\ \top\ \bot) \\
& = & (\mathtt{f}^{\#}\ \top\ \infty) \sqcup (\mathtt{f}^{\#}\ \top\ \bot) \\
& = & \mathtt{f}^{\#}\ \top\ \infty
\end{array}$$

A similar derivation will give us the definition for the abstract value of $\mathtt{h}^{\#}\ \bot_{\in}$. The case for the argument value $\top_{\in}$ is somewhat different. We don't know which pattern match will succeed, so we have to take both possibilities into account. This explains why we need the abstract value of the right-hand side `a` of the `nil`-case here.

A nice example to illustrate this method is the `sum` function from Wadler [14]. Its definition is

```
sum nil         = 0
sum (cons x xs) = x + sum xs
```

The result of this function is an integer, so the abstract values that we shall derive momentarily must be in the domain **2**. Furthermore, there is a recursive call in the `cons`-case which will require a fixpoint iteration to find an approximation of the abstract value. This iteration will be omitted; it is similar to the one discussed in Section 3.1. We first consider $\mathtt{sum}^{\#}\ \bot$.

$$\mathtt{sum}^{\#}\ \bot = \bot$$

We now move on to $\mathtt{sum}^{\#}\ \infty$. In the definition of `sum` for the `cons`-case, the function `f` that is applied to `x` and `xs` is `x + sum xs`. Its abstract value thus is $\mathtt{x}^{\#}\ \sqcap\ (\mathtt{sum}^{\#}\ \mathtt{xs}^{\#})$. Since $\mathtt{x}^{\#} = \top$ and $\mathtt{xs}^{\#} = \infty$ we obtain

$$\begin{array}{lcl}
\mathtt{sum}^{\#}\ \infty & = & \top \sqcap (\mathtt{sum}^{\#}\ \infty) \\
& = & \mathtt{sum}^{\#}\ \infty \\
& = & \bot \text{ (after fixpoint iteration)}
\end{array}$$

For $\texttt{sum}^\# \perp_\in$ we find

$$\begin{array}{rcl} \texttt{sum}^\# \perp_\in & = & (\perp \sqcap (\texttt{sum}^\# \top_\in)) \sqcup (\top \sqcap (\texttt{sum}^\# \perp_\in)) \\ & = & \perp \sqcup (\top \sqcap (\texttt{sum}^\# \perp_\in)) \\ & = & \top \sqcap (\texttt{sum}^\# \perp_\in) \\ & = & \texttt{sum}^\# \perp_\in \\ & = & \perp_\in \text{ (after fixpoint iteration)} \end{array}$$

Finally, for $\texttt{sum}^\# \top_\in$ we need the abstract value of the `nil`-case. This is $0^\# = \top$, so we find without much trouble

$$\begin{array}{rcl} \texttt{sum}^\# \top_\in & = & \top \sqcup (\top \sqcap (\texttt{sum}^\# \top)) \\ & = & \top \sqcup \top \\ & = & \top \end{array}$$

So, combining the results that we obtained above, we get the following definition of the abstract value $\texttt{sum}^\# \texttt{x}^\# \texttt{xs}^\#$.

$$\begin{array}{rcl} \texttt{sum}^\# \top_\in & = & \top \\ \texttt{sum}^\# \perp_\in & = & \perp \\ \texttt{sum}^\# \infty & = & \perp \\ \texttt{sum}^\# \perp & = & \perp \end{array}$$

This is indeed what we would expect for the strictness of `sum`. To compute the sum of all elements of a list of integers, each element must be evaluated. If at least one of the elements is a non-terminating calculation, or if the list is infinite, the computation of the sum of all elements will not terminate. If the input is a non-terminating calculation, the computation will not terminate either.

To abstract over lists of any type we need to modify the abstract domain and the definition of the function $\texttt{cons}^\#$. If the elements of a list come from an abstract domain $\mathbb{D}$, we can create the abstract domain $\mathbb{D}^*$ for this list in two steps. First, for every abstract value $d^\# \in \mathbb{D}$ we create a value $d^\#_\in$ in the domain $\mathbb{D}^*$. Then we add the elements $\perp$ and ∞ to this domain. The ordering of the elements in $\mathbb{D}^*$ is

$$\perp \leq \infty \leq \perp_\mathbb{D} \leq \in_\mathbb{D} \leq \top_\mathbb{D}$$

In this ordering $\perp_\mathbb{D}$ and $\top_\mathbb{D}$ are the values constructed from the bottom and top elements of $\mathbb{D}$ (i.e., they are the bottom and top elements of $\mathbb{D}$ appended with the subscript $_\in$) and $\in_\mathbb{D}$ represents all other values constructed from elements of $\mathbb{D}$, with the same ordering as their counterparts in $\mathbb{D}$. For instance, for the domain $\mathbf{2}^*$ we find the elements $\perp_\in$ and $\top_\in$ in the first step.

After adding the elements $\perp$ and ∞ in the second step, we find that $\mathbf{2}^* = \{\perp, \infty, \perp_\in, \top_\in\}$. The ordering of this domain is $\perp \leq \infty \leq \perp_\in \leq \top_\in$. This is indeed the abstract domain that we introduced for lists of integers at the beginning of this section. Similarly, we can construct the abstract domain $\mathbf{2}^{**}$ for lists of lists of integers from $\mathbf{2}^*$; we find that $\mathbf{2}^{**} = \{\perp, \infty, \perp_\in, \infty_\in, \perp_{\in\in}, \top_{\in\in}\}$ with the ordering $\perp \leq \infty \leq \perp_\in \leq \infty_\in \leq \perp_{\in\in} \leq \top_{\in\in}$. We still need to define the function $\texttt{cons}^\# \texttt{x}^\# \texttt{xs}^\#$ for this domain. The three rules below guide us here.

$$\begin{array}{rcl} \texttt{cons}^\# \texttt{x}^\# (\texttt{y}^\#_\in) & = & (\texttt{x}^\# \sqcap \texttt{y}^\#)_\in \\ \texttt{cons}^\# \texttt{x}^\# \infty & = & \infty \\ \texttt{cons}^\# \texttt{x}^\# \perp & = & \infty \end{array}$$

For example, according to the first rule $\texttt{cons}^{\#}\ \bot_{\in}\ \top_{\in\in} = (\bot_{\in} \sqcap \top_{\in})_{\in} = \bot_{\in\in}$, and $\texttt{cons}^{\#}\ \infty\ \bot_{\in\in} = (\infty \sqcap \bot_{\in})_{\in} = \infty_{\in}$. We can find $\texttt{cons}^{\#}$ for the other 46 combinations of $\texttt{x}^{\#} \in \mathbf{2}^{*}$ and $\texttt{xs}^{\#} \in \mathbf{2}^{**}$ in a similar way.

After constructing the proper abstract domain $\mathbb{D}^{*\#}$ for the non-flat domain $\mathbb{D}^{*}$ and defining $\texttt{cons}^{\#}\ \texttt{x}^{\#}\ \texttt{xs}^{\#}$ for $\mathbb{D}^{*\#}$, we can proceed in the same manner as described before, when we discussed lists of integers.

3.4 Related work

Mycroft [8] was the first to consider strictness analysis, although he did not coin the term. He also shows how to use abstract interpretation to perform strictness analysis on flat, first-order domains. Burn *et al.* [1] extended this technique to handle higher-order domains and they give an extensive theoretical proof of the soundness and safeness of their technique. Abstract interpretation on lists as an instance of non-flat domains was first demonstrated by Wadler [14]. Although a more general method for data structures is not presented, Wadler claims that such an extension is not difficult to come up with.

A similar discussion of strictness analysis using abstract interpretation can be found in Chapter 7 of the textbook *Functional Programming and Parallel Graph Rewriting* by Plasmeijer and Van Eekelen [12]. They discuss an additional technique for strictness analysis using abstract reduction, based on the work of Nocker [10].

Conclusion

In a lazy language it can be advantageous to evaluate expressions eagerly. However, changing the evaluation order is in general not safe: it may affect the termination behaviour of the program. We therefore need to perform strictness analysis to determine for which expressions we can safely change the evaluation order. A function is strict in one of its arguments, if evaluating this argument at the call-site of the function does not change the termination behaviour of the function. We have seen how we can use abstract interpretation to determine the strictness of functions on flat domains and of functions on lists of an arbitrary type.

Chapter 4

Strictness analysis with constraints: the top-down way

Introduction

Although Burn *et al.*'s extension of Mycroft's technique for strictness analysis is theoretically sound and safe, it turns out to be impractical since it relies on higher-order abstract functions, which can be expensive. We can use a more elegant approach if we use information about the regular type of expressions. It is reasonable to expect that this information is available to us: type inference is usually performed by a compiler before strictness analysis takes place. For the remainder of this chapter we shall therefore assume all expressions to be well-typed.

The strictness inference algorithm that we discuss here is due to Glynn *et. al* [3]. An important property of this algorithm is *polyvariance*; different strictness information may be derived for separarate applications of the same function. This is similar to polymorphism in the regular type system, where different types may be found for applications of a polymorphic function. The algorithm can also cope with recursive data types, which are not discussed here since they complicate things considerably.

Unfortunately the syntax of the underlying language UL is somewhat different from the syntax given in Section 2. In this chapter we use the UL syntax, which is given in Section 4.1. Additional information on the subject of this chapter can be found in the aforementioned article of Glynn *et al.*

4.1 Preliminaries

In this chapter and the next we shall use a different representation of the strictness of a function, namely by means of a *constrained strictness type*. Such a type consists of the pair (C, τ), where τ is the actual strictness type and C is a constraint that restricts the free variables of τ. The algorithm described in this chapter generates the pair (C, τ) for a given well-typed expression e of type t. For the remainder of this paper, $e :: t$ denotes that expression e is of type t; the notation $e : \tau$ denotes that e is of strictness type τ. The constraint can then

be solved using an appropriate solving algorithm to determine the strictness behaviour of e.

Here is the underlying language that we consider throughout this chapter.

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{var} :: \mathit{typescheme} \\
& | & (\mathit{expr}\ \mathit{expr}) :: \mathit{typescheme} \\
& | & (\text{'}\lambda\text{'}\ \mathit{var}\ \text{'}\rightarrow\text{'}\ \mathit{expr}) :: \mathit{typescheme} \\
& | & (\texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr}) :: \mathit{typescheme} \\
& | & (\texttt{let}\ \mathit{var}\ \texttt{in}\ \mathit{expr}) :: \mathit{typescheme} \\
& | & (\texttt{fix}\ \mathit{var}\ \texttt{in}\ \mathit{expr}) :: \mathit{typescheme} \\
\sigma & ::= & t \\
& | & \forall \overline{\alpha}.t \\
t & ::= & \alpha \\
& | & \texttt{Bool} \\
& | & \texttt{Int} \\
& | & t \rightarrow t
\end{array}
$$

All expressions are explicitly typed with a type scheme. An application expects only one argument, although it is of course allowed to nest applications. Similarly, a lambda abstraction binds exactly one variable and a let expression has exactly one let definition. Furthermore, recursion is made explicit by using a fix construct. Note that types are limited to type variables, the primitive types `Bool` and `Int`, and the arrow type. Extending the type language with additional primitive types is straightforward.

Strictness types can be defined in a similar way as the regular types. We have the constant strictness value $\bot$, strictness variables δ, and the arrow type. strictness schemes are similar to regular type schemes, but have an additional constraint component C. The type scheme $\forall \overline{\delta}.C \Rightarrow \tau$ is equivalent to the constrained type (C, τ). We usually use the variable η to refer to a strictness scheme.

$$
\begin{array}{lcl}
\eta & ::= & \forall \overline{\delta}.C \Rightarrow \tau \\
\tau & ::= & \delta \\
& | & \bot \\
& | & \tau \rightarrow \tau
\end{array}
$$

It is quite straightforward to determine the strictness type of a well-typed expression $e :: t$ by examining the shape (i.e., the structure) of e's regular type t. We simply replace all primitive types by fresh strictness variables and preserve the arrow types. If we define the judgment $t \vdash \tau$ as 'the strictness type τ has the same shape as the type t', we can apply the following rules to find τ given t.

$$(\mathrm{Bool}_{shape})\frac{\delta \text{ is fresh}}{\texttt{Bool} \vdash \delta}$$

$$(\mathrm{Int}_{shape})\frac{\delta \text{ is fresh}}{\texttt{Int} \vdash \delta}$$

$$(\rightarrow_{shape})\frac{t_1 \vdash \tau_1 \quad t_2 \vdash \tau_2}{t_1 \rightarrow t_2 \vdash \tau_1 \rightarrow \tau_2}$$

So finding the strictness type τ is the easy part. But we also need to find the constraint that restrict the free variables of this type. The language for constraints is given by the following syntax.

$$\begin{array}{rcl} C & ::= & \bot \\ & | & \top \\ & | & \delta \\ & | & \delta_1 \supset \ldots \supset \delta_n \\ & | & C \wedge C \\ & | & \exists \overline{\delta}.C \end{array}$$

Here $\supset$ is logical implication. Constraints of this language are all in the Horn family of Boolean formulae. These can be solved using, for example, the algorithm described in Chapter 6.

In the rules of Section 4.2 we shall see structural constraints of the form $\tau_1 \leq_s \tau_2$. Such a constraint expresses that the strictness type τ_1 of a certain expression e_1 subsumes the strictness type τ_2 of another expression e_2 i.e., e_1 is at least as strict as e_2. A structural constraint thus describes a dataflow between these expressions; the strictness type τ_1 of e_1 *implies* the strictness type τ_2 of e_2. We can decompose a structural constraint into a constraint of the above language as follows.

$$\begin{array}{c} [\![(\tau_1 \to \tau_2 \leq_s \tau_3 \to \tau_4)]\!] = [\![(\tau_2 \leq_s \tau_4)]\!] \wedge [\![(\tau_3 \leq_s \tau_1)]\!] \\ [\![(\delta_1 \leq_s \delta_2)]\!] = \delta_1 \supset \delta_2 \end{array}$$

For instance, the constraint $\top \to \bot \leq_s \bot \to \top$ holds, since it can be decomposed and simplified to $\bot \leq_s \top$. This is what we would expect. If the result of a function is non-terminating, it is safe to forget this information. On the other hand, if we can pass any value as an argument to a function, we can provide a non-terminating calculation as the function's argument.

Generalization over a constrained strictness type (C, τ) where the environment Γ is defined as

$$\mathit{generalize}(C, \Gamma, \tau) = (\exists \overline{\delta}.C, \forall \overline{\delta}.C \Rightarrow \tau)$$

where $\overline{\delta} = \mathit{fv}(C, \tau) \backslash \mathit{fv}(\Gamma)$. The constraint C is pushed into the strictness scheme, universally quantifying the strictness variables that only appear in intermediate results. The same strictness variables are discarded from the constraint C that is passed downwards.

4.2 Strictness inference

We formulate the strictness inference algorithm in terms of inference rules that involve judgements of the form

$$\Gamma, e :: t \vdash^{\mathrm{TD}}_{\mathit{strict}} (C, \tau)$$

The set Γ is an environment that contains assumptions about the strictness type of variables. The rules are applied top-down, and Γ is passed in the same

direction (as the reader may have guessed, the annotation 'TD' indeed stands for 'top-down'). The result of the algorithm is the constrained strictness type (C, τ) of the well-typed expression $e :: t$.

We return to our type inference example program from Section 2.2, now with type annotation, as a running example throughout this section.

```
let id = \x -> x
in  \x -> if x == 1 then x else id (id x) :: Int
```

It is assumed that the initial environment Γ_1 contains assumptions about the strictness of the predefined values `(==)` and `1`. Since there is no inference rule for literal expressions, we will consider `1` as a variable. Then $\Gamma_1 = \{$`(==)` : $\forall\overline{\delta}.\{\delta_x \supset \delta_y \supset \delta_z\} \Rightarrow \delta_x \to \delta_y \to \delta_z$, `1` : $\forall\delta_c.\{\delta_c\} \Rightarrow \delta_c\}$, since `(==)` is strict in both arguments and the value `1` can always be evaluated.

In addition to the syntax-directed rules that will be discussed shortly there is an additional rule ($\exists\text{Intro}^{\text{TD}}_{strict}$) that can be used to discard strictness variables that occur in the constraint component of the strictness type, but don't occur in the type component and the environment Γ. This rule may be applied at any time during inference.

$$(\exists\text{Intro}^{\text{TD}}_{strict})\frac{\Gamma, e :: t \vdash^{\text{TD}}_{strict} (C, \tau) \quad \overline{\delta} = fv(C) \backslash fv(\Gamma, \tau)}{\Gamma, e :: t \vdash^{\text{TD}}_{strict} (\exists\overline{\delta}.C, \tau)}$$

As an example, consider the following constraint

$$C = \begin{array}{l} \underline{\delta_2} \supset \underline{\delta_5} \wedge \underline{\delta_1} \supset \underline{\delta_6} \wedge \delta_{10} \supset \underline{\delta_7} \wedge \underline{\delta_8} \supset \delta_{12} \wedge \underline{\delta_3} \supset \delta_{10} \wedge \\ \underline{\delta_3} \supset \delta_9 \wedge \underline{\delta_1} \supset \underline{\delta_2} \supset \delta_{11} \wedge \delta_9 \supset \delta_{11} \supset \underline{\delta_4} \wedge \delta_9 \supset \delta_{12} \supset \underline{\delta_4} \end{array}$$

where the underlined strictness variables (i.e., the variables with an index less than 9) are assumed to occur in the strictness type or the environment at this point. Application of the rule ($\exists\text{Intro}^{\text{TD}}_{strict}$) discards the other strictness variables. This is done by expressing these variables in terms of the variables that should be kept. For the subconstraint $\delta_9 \supset \delta_{11} \supset \delta_4$ this can be done by replacing δ_9 with δ_3 (which follows from the subconstraint $\delta_3 \supset \delta_9$) and replacing δ_{11} with $\delta_1 \supset \delta_2$ (which follows from the subconstraint $\delta_1 \supset \delta_2 \supset \delta_{11}$). This results in the new subconstraint $\delta_3 \supset \delta_1 \supset \delta_2 \supset \delta_4$. Applying the same reduction strategy to the other variables that should be discarded yields

$$C = \left\{ \begin{array}{l} \delta_2 \supset \delta_5 \wedge \delta_1 \supset \delta_6, \delta_3 \supset \delta_7 \wedge \\ \delta_3 \supset \delta_1 \supset \delta_2 \supset \delta_4 \wedge \delta_3 \supset \delta_8 \supset \delta_4 \end{array} \right\}$$

This way the size of the constraint that is passed upwards can be reduced. In our running example we will not apply this rule, however, so the effects of the application of the other rules are easier to see.

For variables we distinguish between lambda-bound variables and other variables; the latter may have a polymorphic strictness type. This type can be inferred by application of the rule ($\text{Var}^{\text{TD}}_{strict}$).

$$(\text{Var}^{\text{TD}}_{strict})\frac{(x : \forall\overline{\delta}.C \Rightarrow \tau) \in \Gamma \quad \overline{\delta'} \text{ new}}{\Gamma, x :: t \vdash^{\text{TD}}_{strict} (\llbracket[\overline{\delta'}/\overline{\delta}]C\rrbracket, [\overline{\delta'}/\overline{\delta}]\tau)}$$

For lambda-bound variables the rule (Var-$\lambda^{\mathrm{TD}}_{strict}$) should be applied.

$$(\text{Var-}\lambda^{\mathrm{TD}}_{strict})\frac{(x:\tau)\in\Gamma \quad t\vdash\tau' \quad C=[\![\tau\leq_s\tau']\!]}{\Gamma, x::t\vdash^{\mathrm{TD}}_{strict}(C,\tau')}$$

The application rule tells us that the type of the argument and the type of the result should match the type of the function.

$$(\mathrm{App}^{\mathrm{TD}}_{strict})\frac{\Gamma, f::t_1\vdash^{\mathrm{TD}}_{strict}(C_1,\tau_1) \quad \Gamma, a::t_2\vdash^{\mathrm{TD}}_{strict}(C_2,\tau_2) \quad t_3\vdash\tau_3 \quad C=C_1\wedge C_2\wedge[\![\tau_1\leq_s\tau_2\to\tau_3]\!]}{\Gamma,((f::t_1)(a::t_2))::t_3\vdash^{\mathrm{TD}}_{strict}(C,\tau_3)}$$

Since all expressions are well-typed, τ_1 has the same shape as $\tau_2\to\tau_3$.

With the rules above we can derive the inference tree in Figure 4.1 for the subexpression `x == 1`. Since `x` is a lambda-bound variable here, we apply the rule (Var-$\lambda^{\mathrm{TD}}_{strict}$) to it. Here it is assumed that the environment Γ_9 contains the assumption $(\texttt{x}:\delta_4)$. We shall see momentarily that this assumption is indeed generated by application of the rule for lambda abstraction higher up in the tree.

Strictness information for the variables `(==)` and `1` can be found in the initial environment. The strictness types and constraints for these variables are obtained by instantiating the corresponding strictness schemes from the environment. For `(==)` the substitution $[\delta_{8a}/\delta_x,\delta_{8b}/\delta_y,\delta_{8c}/\delta_z]$ is applied to the constraint component and type component of the strictness scheme. Likewise, the substitution $[\delta_{10}/\delta_c]$ is applied to the strictness scheme that corresponds to the value `1`.

As we have seen in Chapter 3, a conditional expression certainly terminates if the guard expression and both branches terminate. We conjoin the corresponding constraint with the constraints from the subexpressions.

$$(\mathrm{If}^{\mathrm{TD}}_{strict})\frac{\Gamma, e::\texttt{Bool}\vdash^{\mathrm{TD}}_{strict}(C_0,\delta) \quad t\vdash\tau \quad \Gamma, e_1::t\vdash^{\mathrm{TD}}_{strict}(C_1,\tau_1) \quad \Gamma, e_2::t\vdash^{\mathrm{TD}}_{strict}(C_2,\tau_2) \quad C=C_0\wedge C_1\wedge C_2\wedge(\delta\supset[\![(\tau_1\leq_s\tau)\wedge(\tau_2\leq_s\tau)]\!])}{\Gamma,(\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2::t)\vdash^{\mathrm{TD}}_{strict}(C,\tau)}$$

A lambda abstraction doesn't generate new constraints. We can compute the strictness type of the bound variable from the shape of its type. This information is added to the environment of the body before we infer its strictness type.

$$(\mathrm{Abs}^{\mathrm{TD}}_{strict})\frac{t_1\vdash\tau_1 \quad \Gamma\backslash x.x:\tau_1, e::t_2\vdash^{\mathrm{TD}}_{strict}(C,\tau_2)}{\Gamma,(\lambda x\to e)::t_1\to t_2\vdash^{\mathrm{TD}}_{strict}(C,\tau_1\to\tau_2)}$$

In Figure 4.2 we see the application of the two previous rules to a part of our example program. The lambda-bound variable `x` and its strictness type δ_4 are passed downwards in the environment Γ_5 so every use of this variable in its scope can be related to the strictness type δ_4.

$$\frac{\boxed{\Delta_0} \quad \dfrac{(\texttt{1} : \forall\delta_c.\{\delta_c\} \Rightarrow \delta_c) \in \Gamma_{10}}{\Gamma_{10}, \texttt{1} :: \texttt{Int} \vdash^{\text{TD}}_{strict} (C_{10}, \delta_{10})}\ (\text{Var})}{\Gamma_6, \texttt{x == 1} :: \texttt{Bool} \vdash^{\text{TD}}_{strict} (C_6, \delta_6)}\ (\text{App})$$

where Δ_0 is

$$\frac{\dfrac{(\texttt{(==)} : \forall\overline{\delta}.\{\delta_x \supset \delta_y \supset \delta_z\} \Rightarrow \delta_x \to \delta_y \to \delta_z) \in \Gamma_8}{\Gamma_8, \texttt{(==)} :: \texttt{Int} \to \texttt{Int} \to \texttt{Bool} \vdash^{\text{TD}}_{strict} (C_8, \delta_{8a} \to \delta_{8b} \to \delta_{8c})}\ (\text{Var}) \quad \boxed{\Delta_1}}{\Gamma_7, \texttt{(==) x} :: \texttt{Int} \to \texttt{Bool} \vdash^{\text{TD}}_{strict} (C_7, \delta_{7a} \to \delta_{7b})}\ (\text{App})$$

and Δ_1 is

$$\frac{(\texttt{x} : \delta_4) \in \Gamma_9}{\Gamma_9, \texttt{x} :: \texttt{Int} \vdash^{\text{TD}}_{strict} (C_9, \delta_9)}\ (\text{Var-}\lambda)$$

$\Gamma_6 = \Gamma_5$
$\Gamma_7 = \Gamma_6$
$\Gamma_8 = \Gamma_7$
$\Gamma_9 = \Gamma_7$
$\Gamma_{10} = \Gamma_6$
$C_6 = C_7 \wedge C_{10} \wedge [\![\delta_{7a} \to \delta_{7b} \leq_s \delta_{10} \to \delta_6]\!]$
$C_7 = C_8 \wedge C_9 \wedge [\![\delta_{8a} \to \delta_{8b} \to \delta_{8c} \leq_s \delta_9 \to \delta_{7a} \to \delta_{7b}]\!]$
$C_8 = \delta_{8a} \supset \delta_{8b} \supset \delta_{8c}$
$C_9 = [\![\delta_4 \leq_s \delta_9]\!]$
$C_{10} = \delta_{10}$

Figure 4.1: Inference tree for the subexpression `x == 1`

$$\frac{\dfrac{\boxed{\Delta_2} \quad \dfrac{(\texttt{x} : \delta_4) \in \Gamma_{11}}{\Gamma_{11}, \texttt{x} :: \texttt{Int} \vdash^{\text{TD}}_{strict} (C_{11}, \delta_{11})}\ (\text{Var-}\lambda) \quad \boxed{\Delta_3}}{\Gamma_5, \texttt{if x == 1 then ...} :: \texttt{Int} \vdash^{\text{TD}}_{strict} (C_5, \delta_5)}\ (\text{If})}{\Gamma_4, \lambda\texttt{x -> if x == 1 ...} :: \texttt{Int} \to \texttt{Int} \vdash^{\text{TD}}_{strict} (C_3, \delta_4 \to \delta_5)}\ (\text{Abs})$$

where Δ_2 is the subtree from Figure 4.1 and Δ_3 is the subtree from Figure 4.4

$\Gamma_4 = \Gamma_1 \cup \{\texttt{id} : \forall\delta.\{[\![\delta_2 \leq_s \delta_3]\!]\} \Rightarrow \delta_2 \to \delta_3\}$
$\Gamma_5 = \Gamma_4 \cup \{\texttt{x} : \delta_4\}$
$\Gamma_{11} = \Gamma_5$
$C_4 = C_5$
$C_5 = C_6 \wedge C_{11} \wedge C_{12} \wedge (\delta_6 \supset [\![(\delta_{11} \leq_s \delta_5]\!]) \wedge (\delta_{12} \leq_s \delta_5))$
$C_{11} = [\![\delta_4 \leq_s \delta_{11}]\!]$

Figure 4.2: Inference tree for the subexpression λ`x -> if x == 1 then x else id (id x)`

$$\dfrac{\dfrac{\dfrac{(\texttt{x} : \delta_2) \in \Gamma_3}{\Gamma_3, \texttt{x} :: \forall a.a \vdash^{\mathrm{TD}}_{strict} (C_3, \delta_3)}\ (\text{Var-}\lambda)}{\Gamma_2, \texttt{id = } \lambda\texttt{x -> x} :: \forall a.a \to a \vdash^{\mathrm{TD}}_{strict} (C_2, \delta_2 \to \delta_3)}\ (\text{Abs}) \quad \boxed{\Delta_4}}{\Gamma_1, \texttt{let id = } \lambda\texttt{x in ...} :: \texttt{Int} \to \texttt{Int} \vdash^{\mathrm{TD}}_{strict} (C_1, \delta_4 \to \delta_5)}\ (\text{Let})$$

where Δ_4 is the subtree from Figure 4.2

$\Gamma_1 = \{\texttt{(==)} : \forall\overline{\delta}.\{\delta_x \supset \delta_y \supset \delta_z\} \Rightarrow \delta_x \to \delta_y \to \delta_z, \texttt{1} : \forall\delta_c.\{\delta_c\} \Rightarrow \delta_c\}$
$\Gamma_2 = \Gamma_1$
$\Gamma_3 = \Gamma_2 \cup \{\texttt{x} : \delta_2\}$
$C_1 = (\exists\overline{\delta}[\![\delta_2 \leq_s \delta_3]\!]) \wedge C_4$
$C_2 = C_3$
$C_3 = [\![\delta_2 \leq_s \delta_3]\!]$

Figure 4.3: Inference tree for the expression `let id = λx -> x in λx -> if x == 1 then x else id (id x)`

For a let expression the strictness scheme that is obtained by generalizing the strictness type of the let definition is passed to the body. The constraint from the body is conjoined with the generalized constraint of the let definition.

$$(\text{Let}^{\mathrm{TD}}_{strict})\dfrac{\begin{array}{c}\Gamma\backslash x, e_1 :: t_1 \vdash^{\mathrm{TD}}_{strict} (C_1, \tau_1) \\ generalize(C_1, \Gamma\backslash x, \tau_1) = (C_0, \eta_0) \\ \Gamma\backslash x.x : \eta_0, e_2 :: t_2 \vdash^{\mathrm{TD}}_{strict} (C_2, \tau_2) \\ C = C_0 \wedge C_2\end{array}}{\Gamma, (\texttt{let } x = (e_1 :: t_1) \texttt{ in } e_2) :: t_2 \vdash^{\mathrm{TD}}_{strict} (C, \tau_2)}$$

Note that a fixed order of inference (first the let definition, then the body) is implied by this rule. Figure 4.3 shows the application of this rule to our example program.

To find the strictness type of a fix construct, a fixpoint iteration $\mathcal{F}$ can be performed. This is similar to the fixpoint iteration for recursive functions described in Section 3.1. For a recursive function f, the strictness type of all recursive calls is approximated by $\forall\overline{\delta}.true \Rightarrow \tau$ in the first iteration (where τ is computed from the shape of the type of f) and inference is then performed as usual. This results in the approximation (C, τ) for the strictness type of f.

If C is equal to the constraint set that was used for the approximation of f's strictness type in the previous iteration, we are done. Otherwise, inference is once again performed on the definition of f, using $\forall\overline{\delta}.C \Rightarrow \tau$ as an approximation of the strictness type of f's recursive calls. This is repeated until C remains the same for two successive iterations. Since the domain of possible strictness values of τ is finite, this fixpoint iteration will terminate.

$$(\text{Fix}^{\mathrm{TD}}_{strict})\dfrac{\begin{array}{c}t \vdash \tau \quad \overline{\delta} = fv(\tau) \\ \eta_0 = \forall\overline{\delta}.true \Rightarrow \tau \\ \mathcal{F}(\Gamma\backslash x.x : \eta_0, e :: t) = (C, \tau)\end{array}}{\Gamma, (\texttt{fix } x :: t \texttt{ in } e) :: t \vdash^{\mathrm{TD}}_{strict} (C, \tau)}$$

An alternative approach to handle fix constructs is to assign the same strictness type to all recursive calls. This gives a less accurate result, but is easier to implement and quicker than performing a fixpoint iteration.

$$(\mathrm{FixC}^{\mathrm{TD}}_{strict})\frac{\begin{array}{c} t \vdash \tau' \quad \Gamma \backslash x.x : \tau', e :: t \vdash^{\mathrm{TD}}_{strict} (C_1, \tau) \\ t \vdash \tau'' \\ C = C_1 \wedge [\![\tau \leq_s \tau']\!] \wedge [\![\tau \leq_s \tau'']\!] \end{array}}{\Gamma, (\texttt{fix}\ x :: t\ \texttt{in}\ e) :: t \vdash^{\mathrm{TD}}_{strict} (C, \tau'')}$$

The additional constraint $\tau \leq_s \tau''$ is generated to avoid the introduction of spurious constraints between the arguments to the function.

Polymorphic expressions

So far we have not explicitly dealt with polymorphism of the underlying language. We shall write $(x :: \forall\overline{\alpha}.t) \sharp \bar{t}$ for the instance of the polymorphic variable x that is obtained when the polymorphic type variables $\overline{\alpha}$ are instantiated by the types $\bar{t}$. For example, if we want to apply the polymorphic identity function `id`, defined as `id = \x -> x`, to the function `incrBy1` of type $\texttt{Int} \rightarrow \texttt{Int}$, we write $((\texttt{id} :: \forall\overline{\alpha}.\alpha_1 \rightarrow \alpha_1) \sharp (\texttt{Int} \rightarrow \texttt{Int}))\ (\texttt{incrBy1} :: \texttt{Int} \rightarrow \texttt{Int})$, which means that α_1 is instantiated to $t_1 = \texttt{Int} \rightarrow \texttt{Int}$ for this application of `id`.

The presence of polymorphic application justifies an additional inference rule (Var-Inst$^{\mathrm{TD}}_{strict}$) for variables which, amongst others, instantiates the strictness scheme of a polymorphic variable. For each instance of a polymorphic variable this rule might derive a different strictness type. The deduction system is thus polyvariant.

We follow the convention that variables β refer to strictness variables corresponding to polymorphic variables in UL. For example, the strictness scheme of `id` of type $\forall\alpha_1.\alpha_1 \rightarrow \alpha_1$ is $\forall\overline{\beta}.\beta_{11} \leq_s \beta_{12} \Rightarrow \beta_{11} \rightarrow \beta_{12}$. The strictness variables β_{11} and β_{12} both correspond to the type variable α_1. Note that the names of the polymorphic strictness variables are chosen such that β_{ij} corresponds to the j^{th} occurence of α_i in the regular type. The shape environment Δ is introduced to store the mappings between polymorphic strictness variables and polymorphic type variables. For `id`, $\Delta = \{\beta_{11} : \alpha_1, \beta_{12} : \alpha_1\}$.

Given the shape environment Δ and an instantiated type t_i, the corresponding instantiated strictness types $\overline{\tau}_i$ can be computed. For example, if α_1 is instantiated to $t_1 = \texttt{Int} \rightarrow \texttt{Int}$, and $\Delta = \{\beta_{11} : \alpha_1, \beta_{12} : \alpha_1\}$, the strictness types β_{11} and β_{12} are instantiated to τ_{11} and τ_{12}, respectively. Here τ_{11} and τ_{12} must have the same shape as t_1. This can be achieved by choosing $\tau_{11} = \delta_1 \rightarrow \delta_2$ and $\tau_{12} = \delta_3 \rightarrow \delta_4$, for instance. In the rule (Var-Inst$^{\mathrm{TD}}_{strict}$) below this computation is written as

$$\text{for each } \Delta \vdash \beta_{ij} : \alpha_i \text{ generate } \Delta, t_i \vdash \tau_{ij}$$

By applying the substitution $[\overline{\tau}/\overline{\beta}]$ to the strictness type component τ of the polymorphic variable x and replacing all monomorphic strictness variables $\overline{\delta}$ in τ by fresh ones, τ is instantiated for an instance of x.

The constraints bound by the strictness scheme of x that contain polymorphic strictness variables remain to be instantiated. Such a constraint is either

of the form

$$\delta_1 \supset \ldots \supset \delta_n \supset \beta \supset \beta'$$

or of the form

$$\delta_1 \supset \ldots \supset \delta_n \supset \beta$$

where $0 \leq n$. Constraints of the latter form usually occur when there is no information available about the strictness behaviour of a function, for instance because it has been imported from another module.

Which constraints should be instantiated is in part determined by the *polarity* of the strictness variables in the type component τ. The polarity of a strictness variable β in τ can be computed by the following function.

```
pol τ β =
    case τ of
        (τ1 → τ2) → if (pol τ1 β = Neg) ∨ (pol τ2 β = Pos)
                    then Pos
                    else if (pol τ1 β = Pos) ∨ (pol τ2 β = Neg)
                         then Neg
                         else Undefined
        β'        → if β' = β then Pos else Undefined
        _         → Undefined
```

If the result is *Pos* then β appears in positive position in τ. Likewise, if the result is *Neg* it appears in negative position.

The former kind of constraints can be instantiated using the set comprehension $\mathcal{T}_{\alpha_i}$ below. We assume that a set is represented in Haskell as a list, so predefined list operations such as *map* can be used in the following definitions.

$$\mathcal{T}_{\alpha_i}(\Delta, \tau, \overline{\tau}_i) = \left\{ \delta_1 \supset \ldots \supset \delta_n \supset [\![\tau_{ij} \leq_s \tau_{ik}]\!] \;\middle|\; \begin{array}{l} \delta_1 \supset \ldots \supset \delta_n \supset \beta_{ij} \supset \beta_{ik} \in C, \\ (\beta_{ij} : \alpha_i) \in \Delta, \mathrm{pol}\ \tau\ \beta_{ij} = \mathit{Neg}, \\ (\beta_{ik} : \alpha_i) \in \Delta, \mathrm{pol}\ \tau\ \beta_{ik} = \mathit{Pos}, \\ \mathrm{all}\ (\mathrm{map}\ (\mathrm{pol}\ \tau)\ \{\delta_1, \ldots, \delta_n\}) = \mathit{Neg} \end{array} \right\}$$

The following function $\mathcal{T}'_{\alpha_i}$ instantiates constraints of the latter form. Only the final result of a function needs to be constrained. The function *res* obtains the appropriate strictness type for this result.

$$\mathcal{T}'_{\alpha_i}(\Delta, \tau, \overline{\tau}_i) = \left\{ \delta_1 \supset \ldots \supset \delta_n \supset (\mathrm{res}\ \tau_{ij}) \;\middle|\; \begin{array}{l} \delta_1 \supset \ldots \supset \delta_n \supset \beta_{ij} \in C, \\ (\beta_{ij} : \alpha_i) \in \Delta, \mathrm{pol}\ \tau\ \beta_{ij} = \mathit{Pos}, \\ \mathrm{all}\ (\mathrm{map}\ (\mathrm{pol}\ \tau)\ \{\delta_1, \ldots, \delta_n\}) = \mathit{Neg} \end{array} \right\}$$

Here the definition of the function *res* is as follows.

```
res τ =
    case τ of
        (τ → τ') → res τ'
        β        → β
        δ        → δ
```

The functions $\mathcal{T}_{\alpha_i}$ and $\mathcal{T}'_{\alpha_i}$ are applied to every underlying polymorphic type variable α_i in the range of Δ. We thus obtain the following rule for polymorphic application.

$$(\text{Var-Inst}^{\text{TD}}_{strict})\frac{\begin{array}{c} x : \forall\overline{\beta}, \overline{\delta}.C \Rightarrow \tau \in \mathcal{T} \quad t' = [\overline{t}/\overline{\alpha}]t \quad \tau, t \vdash \Delta \\ \text{for each } \Delta \vdash \beta_{ij} : \alpha_i \text{ generate } \Delta, t_i \vdash \tau_{ij} \\ C' = [\![[\overline{\delta'}/\overline{\delta}]C]\!] \wedge \mathcal{T}(\Delta, \tau, \overline{\tau}) \quad \tau' = [\overline{\tau}/\overline{\beta}, \overline{\delta'}/\overline{\delta}] \quad \text{where } \overline{\delta'} \text{ fresh} \end{array}}{\Gamma, ((x :: \forall\overline{\alpha}.\overline{t})\sharp :: t' \vdash^{\text{TD}}_{strict} (C', \tau')}$$

where

$$\mathcal{T}(\Delta, \tau, \overline{\tau}) = \bigwedge_{\alpha_i \,\in\, ran(\Delta)} (\mathcal{T}_{\alpha_i}(\Delta, \tau, \overline{\tau}_i) \wedge \mathcal{T}'_{\alpha_i}(\Delta, \tau, \overline{\tau}_i))$$

Note that the substitution $[\overline{\delta}'/\overline{\delta}]$ is applied to both the type component τ and the constraint component C.

We can apply the rule (Var-Inst$^{\text{TD}}_{strict}$) to the two instances of the function `id` in the example program. In both cases the type $\forall a.a \to a$ of `id` is instantiated to `Int` $\to$ `Int`. In this case application of this rule is relatively simple: there is only one constraint that should be instantiated (namely $[\![\delta_2 \leq_s \delta_3]\!]$), which contains only two polymorphic strictness variables δ_2 and δ_3 that both correspond to the polymorphic type variable a. The polymorphic strictness variable δ_2 is instantiated to δ_{13a} and δ_{15a}, respectively, for each instance. Similarly, δ_3 is instantiated to δ_{13b} and δ_{15b}, respectively. The result is shown in Figure 4.4.

Conclusion

We can use an efficient constrained-based technique to perform strictness analysis if type information of the underlying program is available to us. We can then determine strictness types for all subexpressions and generate appropriate constraints on these types. In this chapter we have given a top-down strictness inference algorithm, as a system of deduction rules, that accomplishes this for a Haskell-like language with polymorphism and polyvariance.

This algorithm is largely syntax-directed. The constraints that are collected are all in the Horn subset of Boolean formulae, for which efficient solving algorithms are available. Data types are also supported, although these are not discussed here.

$$\dfrac{\dfrac{(\texttt{id} : \forall\delta.\{[\![\delta_2 \leq_s \delta_3]\!]\} \Rightarrow \delta_2 \to \delta_3) \in \Gamma_{13}}{M_1, \Gamma_{13}, \texttt{id} :: \texttt{Int} \to \texttt{Int} \vdash^{\text{TD}}_{strict} (C_{13}, \delta_{13a} \to \delta_{13b})}\ \text{(Var-Inst)} \quad \boxed{\Delta_5}}{\Gamma_{12}, \texttt{id (id x)} :: \texttt{Int} \vdash^{\text{TD}}_{strict} (C_{12}, \delta_{12})}\ \text{(App)}$$

where Δ_5 is

$$\dfrac{\dfrac{(\texttt{id} : \forall\delta.\{[\![\delta_2 \leq_s \delta_3]\!]\} \Rightarrow \delta_2 \to \delta_3) \in \Gamma_{15}}{\Gamma_{15}, \texttt{id} :: \texttt{Int} \to \texttt{Int} \vdash^{\text{TD}}_{strict} (C_{15}, \delta_{15a} \to \delta_{15b})}\ \text{(Var-Inst)} \quad \boxed{\Delta_6}}{\Gamma_{14}, \texttt{id x} :: \texttt{Int} \vdash^{\text{TD}}_{strict} (C_{14}, \delta_{14})}\ \text{(App)}$$

and Δ_6 is

$$\dfrac{(\texttt{x} : \delta_4) \in \Gamma_{16}}{\Gamma_{16}, \texttt{x} :: \texttt{Int} \vdash^{\text{TD}}_{strict} (C_{16}, \delta_{16})}\ \text{(Var-}\lambda\text{)}$$

$\Gamma_{12} = \Gamma_5$
$\Gamma_{13} = \Gamma_{12}$
$\Gamma_{14} = \Gamma_{12}$
$\Gamma_{15} = \Gamma_{14}$
$\Gamma_{16} = \Gamma_{14}$
$C_{12} = C_{13} \wedge C_{14} \wedge [\![\delta_{13a} \to \delta_{13b} \leq_s \delta_{14} \to \delta_{12}]\!]$
$C_{13} = [\![\delta_{13a} \leq_s \delta_{13b}]\!]$
$C_{14} = C_{15} \wedge C_{16} \leq_s [\![\delta_{15a} \to \delta_{15b} \leq_s \delta_{16} \to \delta_{14}]\!]$
$C_{15} = [\![\delta_{15a} \leq_s \delta_{15b}]\!]$
$C_{16} = [\![\delta_4 \leq_s \delta_{16}]\!]$

Figure 4.4: Inference tree for the subexpression `id (id x)`

Chapter 5

Strictness analysis with constraints: the bottom-up way

Introduction

The strictness inference algorithm of Glynn *et al.* is quite similar to the type inference algorithm used to implement a type inferencer for Helium in the Top framework. However, we cannot use this algorithm directly to implement a strictness analysis for Helium in the same framework.

Firstly, the algorithm is top-down oriented. To pinpoint the location where an inconsistency originates from (which is important if we want to inform the user on this) a bottom-up collection of the strictness information is preferred. Secondly, the syntax of the underlying language UL of this algorithm resembles the syntax of Helium, but is more restrictive at some points. Some desugaring would be necessary to translate Helium programs to a core language that fits the syntax of UL. Desugaring, however, is something that should be avoided to keep as close to the conceptual view of the programmer as possible when we provide feedback.

Strictness inference cannot fail in the sense that it discards incorrect programs. It just infers a certain property of a program, namely its strictness behaviour. Additionally, we would like to allow *explicit strictness typing*. The user can then annotate an expression with a strictness type similar to a type annotation in, for instance, Haskell. This strictness type annotation indicates which arguments of a function should be evaluated eagerly, according to the user. This may differ from the strictness behaviour of the function as expected by the strictness checker based on the inferred strictness type and can thus be a source of inconsistencies. This way, explicit strictness typing introduces the need for error reporting in the context of strictness analysis.

In Section 5.2 a variant of Glynn *et al.*'s deduction system that largely overcomes these problems is introduced. The syntax of the underlying language resembles the subset of Helium described in Chapter 2, although it is still a bit more restrictive with respect to let expressions. Quite a few concepts from the

type inference algorithm of Chapter 2 reappear here too. Instance constraints are used to deal with polymorphism, for example. In Section 5.3 we show how to solve these instance constraints. In Section 5.4 the algorithm is extended to cope with explicit strictness typing.

5.1 Preliminaries

In this chapter we consider the same subset of Helium as in Chapter 2 with two slight modifications. Firstly, all expressions are assumed to be explicitly annotated with their type schemes (like in Chapter 4). Secondly, a let expression contains exactly one definition (although nesting of let expressions is allowed).

The constraints that are generated by the algorithm are part of its result, which is again a pair (C, τ) that represents a constrained strictness type. It is convenient for the constraint solving algorithm (see Chapter 6) to collect the constraints from the subexpressions as a set rather than as a conjunction (which results in a single, possibly large, constraint). So, like in Chapter 2, the constraint component C is a set of constraints.

To cope with polymorphism we use *implicit instance constraints* of the form $\tau \leq_M \tau'$ which are similar to the implicit instance constraints for regular types as introduced in Chapter 2. *Explicit instance constraints* of the form $\tau \preceq \eta$ are used for the instances of a function of which the strictness scheme is known, usually because the function is predefined (think of prelude functions).

$$\begin{array}{rcl} C & ::= & \tau \\ & | & \tau \supset \ldots \supset \tau \\ & | & [\![(\tau \leq_M \tau)]\!] \\ & | & [\![(\tau \preceq \eta)]\!] \end{array}$$

In Section 5.3 we shall see how to solve instance constraints.

To derive the strictness type of an expression from its regular type we can use the shape rules of Chapter 4. An additional shape rule for type variables is needed, which is given below.

$$(\mathrm{Var}_{shape})\frac{\delta \text{ is fresh}}{a \vdash \delta}$$

5.2 Strictness inference

The judgements in this section are of the form

$$M, \mathcal{A}, e :: t \vdash^{\mathrm{BU}}_{strict} (C, \tau)$$

Only the set M of monomorphic strictness variables is passed in a top-down fashion. It contains the strictness variables that are associated with variables bound by lambda abstractions, since these are certainly monomorphic. As usual $\mathcal{A}$ is an assumption set. It is more or less similar to the environment Γ from the top-down algorithm, except for the direction in which it is passed.

We shall be conservative concerning optimization of the generated constraints; we don't want to throw away information that seems insignificant

for the final result, but that may be useful to give informative feedback to the user. For example, there is no such rule here like the rule ($\exists\text{Intro}^{\text{TD}}_{strict}$) from the previous chapter. As a fortunate consequence, all rules are syntax-directed.

Once again, we will use the type-annotated program

```
let id = \x -> x
in  \x -> if x == 1 then x else id (id x) :: Int
```

as a running example in this section.

The rules for literals and variables are almost identical to the respective rules of the type inference algorithm of Chapter 2. The shape of the regular type is used to derive the strictness type.

$$(\text{Lit}^{\text{BU}}_{strict})\frac{t \vdash \tau}{M, \emptyset, literal :: t \vdash^{\text{BU}}_{strict} (\{\tau\}, \tau)}$$

$$(\text{Var}^{\text{BU}}_{strict})\frac{t \vdash \tau}{M, \{x :: t\} \vdash^{\text{BU}}_{strict} (\emptyset, \tau)}$$

Since the rules are applied bottom-up, assumptions on the strictness type of a variable are generated at the locations where the variables are used. They are dealt with (i.e., used to generate appropriate constraints) at the binding location of the variable, which is either a lambda abstraction or a let expression. This means that a single rule for variables suffices here, since all we have to do at this point is to generate an assumption. This is not influenced by the type of construct that binds the variable.

The rule for application is a generalization of the rule ($\text{App}^{\text{TD}}_{strict}$) from the previous chapter. Rather than dealing with all arguments to a function one by one in a curried fashion, we consider all arguments to a function at once when the rule ($\text{App}^{\text{BU}}_{strict}$) is applied. The flow of information is the same, but there are no intermediate results that require the generation of additional constraints.

$$(\text{App}^{\text{BU}}_{strict})\frac{\begin{array}{c} M, \mathcal{A}, f :: t \vdash^{\text{BU}}_{strict} (C, \tau) \\ M, \mathcal{A}_i, a_i :: t_i \vdash^{\text{BU}}_{strict} (C_i, \tau_i) \text{ for } 1 \le i \le n \\ t' \vdash \tau' \\ C' = C \cup \bigcup_i C_i \cup \{[\![\tau \le_s \tau_1 \to \ldots \to \tau_n \to \tau']\!]\} \end{array}}{M, \mathcal{A} \cup \bigcup_i \mathcal{A}_i, ((f :: t)(a_1 :: t_1) \ldots (a_n :: t_n)) :: t' \vdash^{\text{BU}}_{strict} (C', \tau')}$$

We can now derive the inference tree in Figure 5.1 for the subexpression `x == 1`, and Figure 5.2 for the subexpression `id (id x)` from the example program.

The rule for conditional expressions is essentially the same as its top-down counterpart.

$$(\text{If}^{\text{BU}}_{strict})\frac{\begin{array}{c} M, \mathcal{A}', e :: \texttt{Bool} \vdash^{\text{BU}}_{strict} (C, \delta) \quad t \vdash \tau \\ M, \mathcal{A}_1, e_1 :: t \vdash^{\text{BU}}_{strict} (C_1, \tau_1) \quad M, \mathcal{A}_2, e_2 :: t \vdash^{\text{BU}}_{strict} (C_2, \tau_2) \\ C' = C \cup C_1 \cup C_2 \cup \{\delta \supset [\![\tau_1 \le_s \tau]\!], \delta \supset [\![\tau_2 \le_s \tau]\!]\} \end{array}}{M, \mathcal{A}' \cup \mathcal{A}_1 \cup \mathcal{A}_2, (\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 :: t) \vdash^{\text{BU}}_{strict} (C', \tau)}$$

$$\dfrac{\dfrac{\texttt{Int} \to \texttt{Int} \to \texttt{Bool} \vdash \delta_{7a} \to \delta_{7b} \to \delta_{7c}}{M_1, \mathcal{A}_7, \texttt{(==)} :: \texttt{Int} \to \texttt{Int} \to \texttt{Bool} \vdash^{\text{BU}}_{strict} (C_7, \delta_{7a} \to \delta_{7b} \to \delta_{7c})}\ (\text{Var}) \quad \boxed{\Delta_0} \quad \boxed{\Delta_1}}{M_1, \mathcal{A}_6, \texttt{x == 1} :: \texttt{Bool} \vdash^{\text{BU}}_{strict} (C_6, \delta_6)}\ (\text{App})$$

where Δ_0 is

$$\dfrac{\texttt{Int} \vdash \delta_8}{M_1, \mathcal{A}_8, \texttt{x} :: \texttt{Int} \vdash^{\text{BU}}_{strict} (C_8, \delta_8)}\ (\text{Var})$$

and Δ_1 is

$$\dfrac{\texttt{Int} \vdash \delta_9}{M_1, \mathcal{A}_9, \texttt{1} :: \texttt{Int} \vdash^{\text{BU}}_{strict} (C_9, \delta_9)}\ (\text{Lit})$$

$\mathcal{A}_6 = \mathcal{A}_7 \cup \mathcal{A}_8 \cup \mathcal{A}_9$
$\mathcal{A}_7 = \{\texttt{(==)} : \delta_{7a} \to \delta_{7b} \to \delta_{7c}\}$
$\mathcal{A}_8 = \{\texttt{x} : \delta_8\}$
$\mathcal{A}_9 = \emptyset$
$C_6 = C_7 \cup C_8 \cup C_9 \cup \{[\![\delta_{7a} \to \delta_{7b} \to \delta_{7c} \leq_s \delta_8 \to \delta_9 \to \delta_6]\!]\}$
$C_7 = \emptyset$
$C_8 = \emptyset$
$C_9 = \{\delta_9\}$

Figure 5.1: Inference tree for the subexpression `x == 1`

In Figure 5.3 the application of this rule to the conditional expression `if x == 1 then x else id (id x)` from the example program is shown.

The rule for lambda abstraction closely resembles the rule (Abs_{type}) from Chapter 2. Rather than introducing a rule for lambda-bound patterns, we can compute the strictness type of each lambda-bound variable from the variable's regular type.

$$(\text{Abs}^{\text{BU}}_{strict})\dfrac{\begin{array}{c}\mathcal{A}'_i = \{p_i : \tau_i \mid p_i :: t_i, t_i \vdash \tau_i\} \text{ for } 1 \leq i \leq n \quad \mathcal{A}' = \bigcup_i \mathcal{A}'_i \\ M \cup ran(\mathcal{A}'), \mathcal{A}, e :: t \vdash^{\text{BU}}_{strict} (C, \tau') \\ t_1 \to \ldots \to t_n \to t \vdash \tau \\ C' = C \cup (\mathcal{A}' \leq_s \mathcal{A}) \cup \{[\![\tau_1 \to \ldots \to \tau_n \to \tau' \leq_s \tau]\!]\}\end{array}}{M, \mathcal{A} \backslash dom(\mathcal{A}'), (\lambda p_1 \ldots p_n \to e) :: t_1 \to \ldots \to t_n \to t \vdash^{\text{BU}}_{strict} (C', \tau)}$$

Figure 5.3 depicts the inference tree of the lambda abstraction in the body of our example program. Here the strictness variable δ_{3d} corresponds to the lambda-bound variable `x` and is calculated from the shape of the type of `x`.

Now consider the rule for let expressions

$$(\text{Let}^{\text{BU}}_{strict})\dfrac{\begin{array}{c}M, \mathcal{A}_1, e_1 :: t_1 \vdash^{\text{BU}}_{strict} (C_1, \tau_1) \quad M, \mathcal{A}_2, e_2 :: t_2 \vdash^{\text{BU}}_{strict} (C_2, \tau_2) \quad t \vdash \tau \\ C = C_1 \cup \{\tau_2 \leq_s \tau\} \cup (\mathcal{A}_1 \leq_M \{x : \tau_1\}) \cup (\mathcal{A}_2 \leq_M \{x : \tau_1\}) \cup C_2\end{array}}{M, (\mathcal{A}_1 \cup \mathcal{A}_2) \backslash x, (\texttt{let } x = (e_1 :: t_1) \texttt{ in } (e_2 :: t_2)) :: t \vdash^{\text{BU}}_{strict} (C, \tau)}$$

The constraints $(\mathcal{A}_1 \leq_M \{x : \tau_1\})$ express that the strictness types of recursive calls in the let definition are approximated by the strictness type of the let definition as a whole. The same constraints that hold for the let definition must

$$\dfrac{\dfrac{\texttt{Int} \to \texttt{Int} \vdash \delta_{12a} \to \delta_{12b}}{M_1, \mathcal{A}_{12}, \texttt{id} :: \texttt{Int} \to \texttt{Int} \vdash^{\text{BU}}_{strict} (C_{12}, \delta_{12a} \to \delta_{12b})}\ (\text{Var}) \quad \boxed{\Delta_2}}{M_1, \mathcal{A}_{11}, \texttt{id (id x)} :: \texttt{Int} \vdash^{\text{BU}}_{strict} (C_{11}, \delta_{11}}\ (\text{App})$$

where Δ_2 is

$$\dfrac{\dfrac{\texttt{Int} \to \texttt{Int} \vdash \delta_{14a} \to \delta_{14b}}{M_1, \mathcal{A}_{14}, \texttt{id} :: \texttt{Int} \to \texttt{Int} \vdash^{\text{BU}}_{strict} (C_{14}, \delta_{14a} \to \delta_{14b})}\ (\text{Var}) \quad \boxed{\Delta_3}}{M_1, \mathcal{A}_{13}, \texttt{id x} :: \texttt{Int} \vdash^{\text{BU}}_{strict} (C_{13}, \delta_{13})}\ (\text{App})$$

and Δ_3 is

$$\dfrac{\texttt{Int} \vdash \delta_{15}}{M_1, \mathcal{A}_{15}, \texttt{x} :: \texttt{Int} \vdash^{\text{BU}}_{strict} (C_{15}, \delta_{15})}\ (\text{Var})$$

$\mathcal{A}_{11} = \mathcal{A}_{12} \cup \mathcal{A}_{13}$
$\mathcal{A}_{12} = \{\texttt{id} : \delta_{12a} \to \delta_{12b}\}$
$\mathcal{A}_{13} = \mathcal{A}_{14} \cup \mathcal{A}_{15}$
$\mathcal{A}_{14} = \{\texttt{id} : \delta_{14a} \to \delta_{14b}\}$
$\mathcal{A}_{15} = \{\texttt{x} : \delta_{15}\}$
$C_{11} = C_{12} \cup C_{13} \cup \{[\![\delta_{12a} \to \delta_{12b} \leq_s \delta_{13} \to \delta_{11}]\!]\}$
$C_{12} = \emptyset$
$C_{13} = C_{14} \cup C_{15} \cup \{[\![\delta_{14a} \to \delta_{14b} \leq_s \delta_{15} \to \delta_{13}]\!]\}$
$C_{14} = \emptyset$
$C_{15} = \emptyset$

Figure 5.2: Inference tree for the subexpression `id (id x)`

$$\dfrac{\boxed{\Delta_4} \quad \dfrac{\texttt{Int} \vdash \delta_{10}}{M_1, \mathcal{A}_{10}, \texttt{x} :: \texttt{Int} \vdash^{\text{BU}}_{strict} (C_{10}, \delta_{10})}\ (\text{Var}) \quad \boxed{\Delta_5}}{M_1, \mathcal{A}_5, \texttt{if x == 1 then ...} :: \texttt{Int} \vdash^{\text{BU}}_{strict} (C_5, \delta_5)}\ (\text{If})}{M_0, \mathcal{A}_3, \lambda\texttt{x -> if x == 1 ...} :: \texttt{Int} \to \texttt{Int} \vdash^{\text{BU}}_{strict} (C_3, \delta_{3a} \to \delta_{3b})}\ (\text{Abs})$$

where Δ_4 is the subtree from Figure 5.1 and Δ_5 is the subtree from Figure 5.2

$\mathcal{A}_3 = \mathcal{A}_5 \backslash \texttt{x} = \mathcal{A}_7 \cup \mathcal{A}_9 \cup \mathcal{A}_{12} \cup \mathcal{A}_{14}$
$\mathcal{A}_5 = \mathcal{A}_6 \cup \mathcal{A}_{10} \cup \mathcal{A}_{11}$
$C_3 = C_5 \cup \{[\![\delta_{3d} \leq_s \delta_8]\!], [\![\delta_{3d} \leq_s \delta_{10}]\!], [\![\delta_{3d} \leq_s \delta_{15}]\!], [\![\delta_{3d} \to \delta_5 \leq_s \delta_{3a} \to \delta_{3b}]\!]\}$
$M_1 = M_0 \cup \{\delta_{3d}\}$
$C_5 = C_6 \cup C_{10} \cup C_{11} \cup \{\delta_6 \supset [\![\delta_{10} \leq_s \delta_5]\!], \delta_6 \supset [\![\delta_{11} \leq_s \delta_5]\!]\}$

Figure 5.3: Inference tree for the subexpression `λx -> if x == 1 then x else id (id x)`

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall a.a \vdash \delta_4}{M_2, \mathcal{A}_4, \mathtt{x} :: \forall a.a \vdash^{\mathrm{BU}}_{strict} (C_4, \delta_4)}\ (\mathrm{Var})}{M_0, \mathcal{A}_2, \mathtt{id = \lambda x \text{ -> } x} :: \forall a.a \to a \vdash^{\mathrm{BU}}_{strict} (C_2, \delta_{2a} \to \delta_{2b})}\ (\mathrm{Abs}) \quad \boxed{\Delta_6}}{M_0, \mathcal{A}_1, \mathtt{let\ id = \lambda x\ in\ ...} :: \mathtt{Int} \to \mathtt{Int} \vdash^{\mathrm{BU}}_{strict} (C_1, \delta_{1a} \to \delta_{1b})}\ (\mathrm{Let})}{\mathcal{P}, M_0, \mathcal{A}_0, \mathtt{let\ id = \lambda x\ in\ ...} :: \mathtt{Int} \to \mathtt{Int} \vdash^{\mathrm{BU}}_{strict} (C_0, \delta_{1a} \to \delta_{1b})}\ (\mathrm{Prel})$$

where Δ_6 is the subtree from Figure 5.3

$$\begin{aligned}
&\mathcal{A}_0 = \mathcal{A}_1 \backslash dom(\mathcal{P}) = \emptyset \\
&\mathcal{A}_1 = (\mathcal{A}_2 \cup \mathcal{A}_3) \backslash \mathtt{id} = \mathcal{A}_2 \cup \mathcal{A}_6 \cup \mathcal{A}_{10} \cup \mathcal{A}_{15} \\
&\mathcal{A}_2 = \mathcal{A}_4 \backslash \mathtt{x} = \emptyset \\
&\mathcal{A}_4 = \{\mathtt{x} : \delta_4\} \\
&C_0 = C_1 \cup \{\delta_{7a} \to \delta_{7b} \to \delta_{7c} \preceq \forall\overline{\beta}.\{\beta_1 \supset \beta_2 \supset \beta_3\} \Rightarrow \beta_1 \to \beta_2 \to \beta_3\} \\
&C_1 = C_2 \cup \{\delta_{12a} \to \delta_{12b} \leq_\emptyset \delta_{2a} \to \delta_{2b}, \delta_{14a} \to \delta_{14b} \leq_\emptyset \delta_{2a} \to \delta_{2b}, \\
&\qquad\qquad [\![\delta_{1a} \to \delta_{1b} \leq_s \delta_{2a} \to \delta_{2b}]\!]\} \cup C_3 \\
&C_2 = C_4 \cup \{[\![\delta_{2c} \leq_s \delta_4]\!], [\![\delta_{2a} \to \delta_{2b} \leq_s \delta_{2c} \to \delta_4]\!]\} \\
&C_4 = \emptyset \\
&M_0 = \emptyset \\
&M_2 = M_0 \cup \{\delta_{2c}\}
\end{aligned}$$

Figure 5.4: Inference tree for the expression `let id = λx -> x in λx -> if x == 1 then x else id (id x)`

hold for its recurive calls, which is enforced by generating implicit instance constraints here (see Section 5.3 for more details). This is a syntax-directed approach, based on the name that is assigned to the let-defined expression. Since recursion is always name-based in Haskell, this approach is conceptually closer to the Haskell syntax than the use of an explicit fixpoint construct.

Furthermore, implicit instance constraints ($\mathcal{A}_2 \leq_M \{x : \tau_1\}$) are generated for all uses of the let-defined expression in the body. This can be seen in Figure 5.4, where the rule is applied to our example program.

Note that a let expression contains exactly one let definition. If we want to allow multiple let definitions, a binding group analysis like the one described in Section 2.2 can be used. We believe that this extension is not difficult to make given the existing binding group analysis for regular type inference. This remains, however, future work.

A syntax-directed strictness inference can be performed using the above rules. However, after this inference is performed, the top-level assumption set may be non-empty because it still contains assumptions about imported definitions. It remains to add constraints on these imported definitions to the final constraint set.

To cope with these definitions, we assume there is an additional assumption set $\mathcal{P}$ available that relates all predefined definitions to a strictness scheme that quantifies the appropriate constraints. For example, the assumption for the operator `(==)` may look like $(\mathtt{(==)} : \forall\overline{\beta}.\{\beta_1 \supset \beta_2 \supset \beta_3\} \Rightarrow \beta_1 \to \beta_2 \to \beta_3\})$, which describes that the operator is strict in both arguments (as usual).

The following rule can then be applied at top-level

$$(\mathrm{Prel}^{\mathrm{BU}}_{strict})\frac{\begin{array}{c} M, \mathcal{A}, (\texttt{let}\ x = (e_1 :: t_1)\ \texttt{in}\ e_2) :: t_2 \vdash^{\mathrm{BU}}_{strict} (C, \tau) \quad \mathcal{P} \\ C' = C \cup (\mathcal{A} \preceq \mathcal{P}) \end{array}}{\mathcal{P}, M, \mathcal{A}, (\texttt{let}\ x = (e_1 :: t_1)\ \texttt{in}\ e_2) :: t_2 \vdash^{\mathrm{BU}}_{strict} (C', \tau)}$$

Application of the rule $(\mathrm{Prel}^{\mathrm{BU}}_{strict})$ concludes the inference process. In Figure 5.4 this rule is applied to generate an instance constraint for the predefined operator `(==)` used in the example program.

5.3 Instantiation

After inference, first the implicit and explicit instance constraints need to be solved by instantiating them. This results in additional basic constraints and subsumption constraints that replace the instance constraints in the constraint set. Throughout this section it is assumed that the constraints over the body of a let expression appear in the constraint set *after* the constraints over the let definition. This requirement can be simply checked by first inferring the strictness of the let definition and adding the hereby obtained constraints before inferring the strictness of the body. We illustrate instantiation by a few examples.

Explicit instance constraints

Consider the following explicit instance constraint.

$$(\delta_0 \to \delta_1) \to (\delta_2 \to \delta_3) \preceq \forall \overline{\delta}.\{\delta_6 \leq_s \delta_7, \delta_6 \to \delta_7 \leq_s \delta_4 \to \delta_5\} \Rightarrow \delta_4 \to \delta_5$$

To make the proper instantiation we should 'match' the strictness type $(\delta_0 \to \delta_1) \to (\delta_2 \to \delta_3)$ with the more general type $\delta_4 \to \delta_5$. We can do so by unifying $\delta_0 \to \delta_1$ with δ_4, and $\delta_2 \to \delta_3$ with δ_5. This results in the substitution $s_1 = [\delta_4 := \delta_8 \to \delta_9,\ \delta_5 := \delta_{10} \to \delta_{11}]$ that instantiates the polymorphic strictness variables δ_4 and δ_5 to fresh strictness types that have the same shape as $\delta_0 \to \delta_1$ and $\delta_2 \to \delta_3$, respectively.

Applying s_1 to the strictness type $\delta_4 \to \delta_5$ yields the instantiated type $(\delta_8 \to \delta_9) \to (\delta_{10} \to \delta_{11})$. We relate this strictness type to $(\delta_0 \to \delta_1) \to (\delta_2 \to \delta_3)$ by adding the constraint $(\delta_8 \to \delta_9) \to (\delta_{10} \to \delta_{11}) \leq_s (\delta_0 \to \delta_1) \to (\delta_2 \to \delta_3)$.

The constraints $\delta_6 \leq_s \delta_7$ and $\delta_6 \to \delta_7 \leq_s \delta_4 \to \delta_5$ need to be instantiated accordingly. After applying s_1 to these constraints, they become $\delta_6 \leq_s \delta_7$ and $\delta_6 \to \delta_7 \leq_s (\delta_8 \to \delta_9) \to (\delta_{10} \to \delta_{11})$, respectively.

Note that another substitution $s_2 = [\delta_6 := \delta_{12} \to \delta_{13},\ \delta_7 := \delta_{14} \to \delta_{15}]$ is required in order to instantiate the strictness variables that occur in the constraints, but not in the strictness type. This also corrects the shape of the left-hand side of the second constraint which should match the shape of the left-hand side. We obtain the instantiated constraints $\delta_{12} \to \delta_{13} \leq_s \delta_{14} \to \delta_{15}$ and $(\delta_{12} \to \delta_{13}) \to (\delta_{14} \to \delta_{15}) \leq_s (\delta_8 \to \delta_9) \to (\delta_{10} \to \delta_{11})$.

The explicit instance constraint above can thus be replaced in the constraint set by

$$\left\{ \begin{array}{l} (\delta_8 \to \delta_9) \to (\delta_{10} \to \delta_{11}) \leq_s (\delta_0 \to \delta_1) \to (\delta_2 \to \delta_3), \\ \delta_{12} \to \delta_{13} \leq_s \delta_{14} \to \delta_{15}, \\ (\delta_{12} \to \delta_{13}) \to (\delta_{14} \to \delta_{15}) \leq_s (\delta_8 \to \delta_9) \to (\delta_{10} \to \delta_{11}) \end{array} \right\}$$

Now suppose that the following explicit instance constraint should be solved.

$$\delta_0 \to \delta_1 \to \delta_2 \preceq \forall\overline{\delta}.\{C \Rightarrow \delta_7 \to \delta_8 \to \delta_9\} \Rightarrow \delta_3 \to \delta_4 \to \delta_5$$

where $C = \{\delta_3 \to \delta_4 \to \delta_5 \preceq \forall\overline{\delta}.\{\delta_7 \supset (\delta_8 \leq_s \delta_9)\}\}$.

The strictness scheme at the left-hand side of this constraint qualifies an additional instance constraint. To properly instantiate the constraint, first this qualified constraint should be instantiated. The procedure for this is the same as described above. Thus $\delta_3 \to \delta_4 \to \delta_5 \preceq \forall\overline{\delta}.\{\delta_7 \supset (\delta_8 \leq_s \delta_9)\} \Rightarrow \delta_7 \to \delta_8 \to \delta_9$ is instantiated to $\{\delta_{10} \to \delta_{11} \to \delta_{12} \leq_s \delta_3 \to \delta_4 \to \delta_5, \delta_{10} \supset (\delta_{11} \leq_s \delta_{12})\}$. We now have that

$$\delta_0 \to \delta_1 \to \delta_2 \preceq \forall\overline{\delta}.C' \Rightarrow \delta_3 \to \delta_4 \to \delta_5$$

where $C' = \{\delta_{10} \to \delta_{11} \to \delta_{12} \leq_s \delta_3 \to \delta_4 \to \delta_5, \delta_{10} \supset (\delta_{11} \leq_s \delta_{12})\}$. This, in turn, can be instantiated to

$$\left\{ \begin{array}{l} \delta_{13} \to \delta_{14} \to \delta_{15} \leq_s \delta_0 \to \delta_1 \to \delta_2, \\ \delta_{16} \to \delta_{17} \to \delta_{18} \leq_s \delta_{13} \to \delta_{14} \to \delta_{15}, \\ \delta_{16} \supset (\delta_{17} \leq_s \delta_{18}) \end{array} \right\}$$

Implicit instance constraints

Solving an implicit instance constraint involves some extra work, since the strictness type occurring at the right-hand side of the constraint should be correctly generalized to a strictness scheme. The—instantiated—constraints that occur in the constraint set before an implicit instance constraint need to be taken into account here. For example, suppose that we have the following constraint set.

$$\left\{ \begin{array}{l} \delta_0 \leq_s \delta_1, \delta_2 \leq_s \delta_3, \delta_3 \leq_s \delta_4, \delta_5 \leq_s \delta_6, \\ \delta_7 \to \delta_8 \leq_{\{\delta_3\}} \delta_4 \to \delta_6 \end{array} \right\}$$

The strictness type $\delta_4 \to \delta_6$ should be generalized to a strictness scheme. Obviously, the constraints $\delta_3 \leq_s \delta_4$ and $\delta_5 \leq_s \delta_6$ should be pushed into this strictness scheme, since these constraints directly involve strictness variables that occur in $\delta_4 \to \delta_6$. Additionally, all constraints that indirectly involve δ_4 and δ_6 should be added to the constraint component of the stricteness scheme, too.

This can be achieved by the following algorithm, given in Haskell-like pseudocode, that takes a set of constraints C and a strictness type τ as its input and returns all constraints in C that should end up in the strictness scheme constructed from τ.

```
getAffectedConstraints C τ =
    let vars = ftv τ
        avars = getAffectedVars C vars
    in { x | x ∈ C, any (`elem` avars) (ftv x) }
```

$$\begin{array}{l}\text{getAffectedVars } C\ vars = (\text{snd } (\text{fixpoint getAffectedVars' } (C,\ vars))) \\ \quad \text{where getAffectedVars' } (C,\ vars) = \\ \qquad\qquad (C,\ vars \cup (\text{ftv } \{\ x \mid x \in C,\ \text{any } (\text{`elem`}\ vars)\ (\text{ftv } x)\ \}))\end{array}$$

Here the overloaded function *ftv* returns all free strictness variables that occur in a strictness type or constraint. The function *fixpoint* is a higher-order function that applies its functional argument recursively to its second argument until a fixpoint is reached. It could be defined as follows.

$$\begin{array}{l}\text{fixpoint } f\ x = \\ \quad \text{let } y = f\ x \\ \quad \text{in if } (y == x) \text{ then } y \text{ else fixpoint } f\ y\end{array}$$

If we apply this procedure to our example, we find that the constraints $\delta_2 \leq_s \delta_3$, $\delta_3 \leq_s \delta_4$, and $\delta_5 \leq_s \delta_6$ should be pushed into the strictness scheme.

Beside the constraints that should be qualified by the strictness scheme, the monomorphic variables should be taken into account when generalization is performed. Note that in the above example the strictness variable δ_3 is monomorphic (it is bound to the '$\leq$' of the implicit instance constraint). However, since $\delta_2 \leq_s \delta_3$ and $\delta_3 \leq_s \delta_4$, the strictness variables δ_2 and δ_4 turn out to be monomorphic too.

The following computation will return all relevant monomorphic variables, given the constraint set C and the original set of monomorphic variables M bound to the implicit instance constraint.

$$\text{getMonoVars } C\ M = (\text{snd } (\text{fixpoint getMonoVars' } (C,\ M)))$$

$$\begin{array}{l}\text{getMonoVars' } (C,\ M) = \\ \quad (C,\ M \cup (\text{concat } (\text{map } (\lambda x \rightarrow \text{getMonoVars"}\ x\ M)\ C)))\end{array}$$

$$\begin{array}{ll}\text{getMonoVars"}\ C\ M = & \\ \quad \text{case } C \text{ of} & \\ \qquad (\tau_1 \rightarrow \tau_2 \leq_s \tau_3 \rightarrow \tau_4) & \rightarrow (\text{getMonoVars"}\ (\tau_3 \leq_s \tau_1)\ M) \cup \\ \qquad\qquad & \quad\ (\text{getMonoVars"}\ (\tau_2 \leq_s \tau_4)\ M) \\ \qquad \delta \leq_s \tau & \rightarrow \text{if } (\delta\ \text{`elem`}\ M) \text{ then } (\text{ftv } \tau) \text{ else } \emptyset \\ \qquad \tau \leq_s \delta & \rightarrow \text{if } (\delta\ \text{`elem`}\ M) \text{ then } (\text{ftv } \tau) \text{ else } \emptyset \\ \qquad \_ & \rightarrow \emptyset\end{array}$$

We define generalization of a strictness type τ given the constraint set C that should be pushed into the strictness scheme and the set of monomorphic variables M as

$$generalize(M, C, \tau) =_{def} \forall \overline{\alpha}.C \Rightarrow \tau \text{ where } \overline{\alpha} = ftv(C) \cup ftv(\tau) \backslash M$$

For our example this yields the strictness scheme

$$\forall \delta_5, \delta_6.\{\delta_2 \leq_s \delta_3, \delta_3 \leq_s \delta_4, \delta_5 \leq_s \delta_6\} \Rightarrow \delta_4 \rightarrow \delta_6$$

Now the strictness type at the right-hand side of the implicit instance constraint has been generalized, this implicit instance constraint can be rewritten to an explicit instance constraint. In our case we obtain

$$\delta_7 \to \delta_8 \preceq \forall \delta_5, \delta_6.\{\delta_2 \leq_s \delta_3, \delta_3 \leq_s \delta_4, \delta_5 \leq_s \delta_6\} \Rightarrow \delta_4 \to \delta_6$$

This explicit instance constraint can now be solved as usual.

5.4 Strictness checking

If we extend the underlying language, we can allow the programmer to document let definitions with strictness annotations. An annotated let definition then looks like this:

```
let f : L -> S
    f = \x y -> ...
in  ...
```

Every strictness value in the strictness annotation (either `S` or `L`) corresponds to an argument to the let-defined function. The value `S` ('strict') represents the strictness value $\bot$ and the value `L` ('lazy') represents the strictness value $\top$. The function `f` above thus is expected to be strict in the second of its two arguments.

Two remarks about this annotation are in place here. Firstly, unlike type annotations, a strictness annotation does not contain a reference to the result of a function, since we cannot assign an expected strictness value to it. Secondly, strictness values can be assigned to functional arguments in their entirety only. Thus we can write strictness annotations such as `apply : S -> L` where the first argument is a function, but a strictness annotation like `apply : (L -> S) -> L` is not allowed. In principal there is no objection to include this kind of strictness annotation, but we are usually only interested in the strictness value of the functional argument as a whole.

For an annotated let definition the user-supplied strictness information has to be compared with the strictness information that is obtained by the strictness inference algorithm. The user-supplied strictness value of each argument is compared to the strictness value found by the inference algorithm. Three outcomes are possible:

1. The strictness value expected by the user is the same as the inferred strictness value. This is the case when the user-supplied value and the derived value are either both $\bot$ (`S`) or both $\top$ (`L`). The user-supplied value is thus consistent with the derived value and no warning message needs to be reported for this argument.

2. The strictness value expected by the user is less precise than the inferred strictness value. This is the case when the user-supplied value is $\top$ and the inferred strictness value is $\bot$. A warning message can be reported for this argument that informs the user that a more precise strictness value was found.

3. The strictness value expected by the user is more precise than the inferred strictness value. This is the case when the user-supplied value is $\bot$ and the inferred strictness value is $\top$. Since the strictness inferencer does not find the most precise strictness information in some cases, it is not certain that the user-supplied information is unsafe. Therefore it is better to report a warning message rather than an error message for this argument.

The above procedure validates the user-supplied information. We can take it one step further and use the strictness annotation as an interface between the let definition and its instances. This is similar to how type annotations are used: it is verified that the explicit type t is consistent with the type derived for the let definition. For every instance of the let-defined expression it is then assumed that it is of type t. Consider the following type-annotated let definition.

```
let id :: Int -> Int
    id = \x -> x
    const = \x y -> x
in  (id const) (id 3) 2
```

The type derived for `id` is $\forall a.a \to a$. The explicit type `Int` $\to$ `Int` is consistent with this type, since it is a specialization of it. All instances of `id` in the body of this expression are now assumed to have type `Int` $\to$ `Int` rather than the more general type $\forall a.a \to a$. The application `(id const)` thus is type incorrect here.

To achieve a similar use of strictness annotations, the strictness information supplied by the programmer must be incorporated in the constraints. First the user-supplied strictness information for a let-defined function `f` is combined with the inferred strictness information. If `f` is expected to be strict in a certain argument, based on at least one of these pieces of information, it is assumed that `f` is indeed strict in this argument. For instance, given the expected strictness information `f : S -> S -> L` and the inferred information `f : S -> L -> L` we assume `f` to be strict in its first argument and not strict in its third argument, based on both pieces of information. Additionally, we assume it to be strict in its second argument, based on the strictness information expected by the programmer.

We can now replace the constraints generated for the let definition of `f` i.e., the constraints C_1 and $(\mathcal{A}_1 \leq_M \{x : \tau_1\})$ generated by the rule $(\text{Let}^{\text{BU}}_{strict})$, by a single constraint of the form $\delta_1 \supset \ldots \supset \delta_n \supset \delta_{n+1}$ that expresses the strictness of `f` in these arguments. Given the strictness type τ inferred for the let definition of `f` we determine which strictness variables correspond to the arguments that `f` is assumed to be strict in. For a functional argument this is the strictness variable that corresponds to its result. For instance, if the strictness type of `f` is $(\delta_1 \to \delta_2) \to \delta_3 \to \delta_4 \to \delta_5$ and `f` is assumed to be strict in its first two arguments, then the strictness variables δ_2 (which corresponds to the result of the functional first argument) and δ_3 (which corresponds to the second argument) are included in the constraint.

Furthermore we need the strictness variable in τ that corresponds to the result of `f` to build the constraint. If τ is $(\delta_1 \to \delta_2) \to \delta_3 \to \delta_4 \to \delta_5$, this is

δ_5. The constraints inferred for the let definition of `f` can then be replaced by the constraint $\delta_2 \supset \delta_3 \supset \delta_5$ to express that `f` is assumed to be strict in its first two arguments. In general, a constraint of the form $\delta_1 \supset \ldots \supset \delta_n \supset \delta_{n+1}$ is generated, where the strictness variables $\delta_1 \ldots \delta_n$ correspond to the arguments that `f` is assumed to be strict in and where δ_{n+1} corresponds to the result of `f`.

A similar approach can be used if we want to generate an assumption set $\mathcal{P}$ for predefined functions from an imported module without actually performing strictness inference on these definitions. We suppose that all function definitions in the module are annotated with a safe strictness annotation and a type annotation. The strictness type τ of a predefined function `p` can then be computed from the provided regular type using the shape rules, as usual. The strictness annotation tells us in which arguments `p` is expected to be strict. Like above, we relate these arguments to the corresponding strictness variables in τ and generate the constraint of the form $C = \delta_1 \supset \ldots \supset \delta_n \supset \delta_{n+1}$ where δ_{n+1} corresponds to the result of `p`. Combining this constraint C with τ we obtain the explicit strictness scheme $\forall\overline{\delta}.C \Rightarrow \tau$ and add the assumption $(\texttt{p} : \forall\overline{\delta}.C \Rightarrow \tau)$ to the assumption set $\mathcal{P}$.

Since the derived strictness scheme is built from the strictness annotation without combining this information with inferred strictness information, we might loose precision here. For instance, if the strictness annotation is `f : S -> S -> L` the function might be strict in its third argument. However, since no strictness inference is performed here, this is not taken into account when the constraint for `f` is generated. It is simply assumed that `f` is not strict in this argument, which is a safe but perhaps less precise assumption. On the other hand, this approach is more time efficient than performing strictness inference on the imported function definitions and might therefore be worth considering.

Conclusion

Combining some of the concepts of Top's type inferencer (a.o. instance constraints and bottom-up collection of information) with some concepts of the strictness inference system of Glynn *et al.* (a.o. polyvariance and constraints in Horn), we can formulate a strictness inference algorithm that fits well into the Top framework and can deal with a subset of the Haskell98 language. The generation of instance constraints requires an additional phase, as compared to the top-down variant of Glynn *et al.*, in which these constraints are reduced to Horn constraints.

With a relatively small extension to this algorithm and the underlying language, the programmer can provide explicit strictness information. This way he can document what strictness behaviour is expected for a let-defined expression. This is then compared to the strictness behaviour that is dervied by the strictness inference algorithm. With an extra extension that generates constraints based on a combination of user-supplied strictness information and inferred strictness information, an explicit strictness annotation can be used as an interface between the let definition of a function and its instances in the body of the underlying let expression.

Chapter 6

Solving the constraints

Introduction

The result of the strictness inference algorithm of Chapter 5 is a pair consisting of a constraint set and a strictness type. In Section 6.1 and Section 6.2 we shall see how we can use this information to determine the strictness behaviour of an expression. In Section 6.3 we discuss how strictness information of intermediate let definitions can be obtained.

6.1 Phased constraint solving

The constraint solving procedure consists of three phases. First we need to rewrite the constraints to a form that can be fed to the consistency checking algorithm. The next phase is an initialization step that extends the constraint set with some additional constraints. In the final phase the consistency checking algorithm determines whether the constraint set is consistent.

Phase 1: Rewriting the constraints

In Section 4.1 we have seen how to translate structural constraints to constraints that are in Horn. Here are the rewrite rules from that section, slightly modified to cope with constraint sets rather than single constraints. This modification is due to the representation of a constrained strictness type's constraint component as a set in Chapter 5.

$$\{[\![(\tau_1 \to \tau_2 \leq_s \tau_3 \to \tau_4)]\!]\} = \{[\![(\tau_2 \leq_s \tau_4)]\!], [\![(\tau_3 \leq_s \tau_1)]\!]\}$$
$$\{[\![(\delta_1 \leq_s \delta_2)]\!]\} = \{\delta_1 \supset \delta_2\}$$

After instantiation of the explicit and implicit instance constraints, all constraints are of a form that can be rewritten to Horn. If we exhaustively apply the rewrite rules, all constraints will be of the form

$$\begin{array}{lcl} H & ::= & \delta \\ & | & \delta_1 \supset \ldots \supset \delta_n \supset \delta \\ & | & \delta_1 \supset \ldots \supset \delta_n \supset \bot \end{array}$$

This is indeed a subset of the language for Horn constraints given in Section 4.1. The strictness variables directly map to positive Boolean literals, which can take on either the Boolean values $\top$ (*true*) or $\bot$ (*false*). Similarly, the strictness value $\bot$ maps to the Boolean value $\bot$ (*false*). We can split a set of Horn clauses in *facts* and *rules*. Facts are all clauses of the form $\bot$ or δ. All other clauses (which are the ones that contain at least one logical implication) are rules.

Phase 2: Initialization

The constraint set C generated by the inference algorithm captures the relation between the strictness values of the various parts of a program. In this phase we use this information to determine how the strictness of arguments to a function influence the strictness of the result of the function.

Recall that in order to examine whether a function is strict in one of its arguments, we must initialize this argument's strictness value to the bottom value of its domain and the strictness values of all other arguments to the top values of their respective domains.

To initialize the strictness value of a function's argument, constraints on the strictness variables occurring in the argument's strictness type are added to the original constraint set C obtained after inference.

- If the argument has a simple strictness type of shape δ its strictness value can be initialized to $\top$ by adding the fact δ to C. To initialize its strictness value to $\bot$ instead, we add the rule $\delta \supset \bot$ to C.

- If the argument has a functional strictness type of shape $\delta_1 \to \ldots \to \delta_n \to \delta$ its strictness value can be initialized to the top value $\lambda\delta_1, \ldots, \delta_n \to \top$ by adding the facts δ and δ_i for $1 \leq i \leq n$. By adding the rule $\delta \supset \bot$ instead of the fact δ, the strictness value is initialized to the bottom value $\lambda\delta_1, \ldots, \delta_n \to \bot$.

The strictness type of the result of the function is initialized to $\bot$ by adding the constraint $\delta_{res} \supset \bot$, where δ_{res} is the strictness variable associated with the result.

As an example, consider the higher-order function f for which the constrained strictness type $(C, (\delta_1 \to \delta_2 \to \delta_3) \to \delta_4 \to \delta_5 \to \delta_6)$ has been derived by the inference algorithm.

- To determine whether f is strict in its first argument, the constraint set $C_1 = C \cup \{\delta_1, \delta_2, \delta_3 \supset \bot, \delta_4, \delta_5, \delta_6 \supset \bot\}$ should be checked for consistency.

- To determine whether f is strict in its second argument, the constraint set $C_2 = C \cup \{\delta_1, \delta_2, \delta_3, \delta_4 \supset \bot, \delta_5, \delta_6 \supset \bot\}$ should be checked for consistency.

- To determine whether f is strict in its third argument, the constraint set $C_3 = C \cup \{\delta_1, \delta_2, \delta_3, \delta_4, \delta_5 \supset \bot, \delta_6 \supset \bot\}$ should be checked for consistency.

For a function g with the constrained strictness type $(C, ((\delta_1 \to \delta_2) \to \delta_3) \to \delta_4 \to \delta_5 \to \delta_6)$ we would have to check the same contraint sets C_1, C_2, and C_3, respectively. This is true, since $(\delta_1 \to \delta_2) \to \delta_3$ is initialized to its bottom value

$\lambda(\delta_1 \to \delta_2) \to \bot$ by initializing the strictness value of its argument to the top value $\lambda\delta_1 \to \top$ and initializing the strictness value of its result to $\bot$. This is achieved by adding the constraints $\{\delta_1, \delta_2, \delta_3 \supset \bot\}$. Similarly, $(\delta_1 \to \delta_2) \to \delta_3$ is initialized to its top value by adding the constraints $\{\delta_1, \delta_2, \delta_3\}$. Thus the constraints that initialize the strictness variables of the functional argument to g are the same as the constraints that initialize the strictness variables of the functional argument to f.

For every argument to a function a new constraint set should be considered. Unfortunately, since we are interested in the strictness of a function in all of its arguments, we need to check all corresponding constraint sets.

Phase 3: Solving the Horn constraints

We now know how to obtain a set of Horn constraints to examine the strictness of a function in one of its arguments. To see whether the function is strict in this argument, we check the consistency of the constraint set.

Each Horn clause has a *head* and a *body*. The head of a fact is simply the fact itself. Its body is empty. The head of a rule is its rightmost atomic value (either a literal or $\bot$) and its body is the part of the rule that implies the head. For instance, the head of the rule $\delta_1 \supset \delta_2 \supset \delta_3 \supset \bot$ is $\bot$ and its body is $\delta_1 \supset \delta_2 \supset \delta_3$.

We can deduce a new fact δ from a set of clauses if this set contains a rule with δ as its head element and all of the atomic values that occur in the body of this rule also occur as facts in the set. We can then add the fact δ to the constraint set. For instance, from the set $\{\delta_1, \delta_2, \delta_3, \delta_4, \delta_1 \supset \delta_2 \supset \delta_3 \supset \delta_5, \delta_4 \supset \delta_5 \supset \bot\}$ we can deduce the facts δ_5 and—after deducing δ_5 and adding it to the constraint set—$\bot$. If we can deduce the fact $\bot$ from the clauses in a set, this set is *inconsistent* (like the set in the above example).

We can formalize this procedure as the following algorithm, written in Haskell-like pseudocode.

```
solve C =
  let f = facts C
      r = rules C
      (f', r') = fixpoint solveStep (f, r)
  in if (⊥ ∈ f') then Inconsistent else Consistent

solveStep (f, r) =
  if (⊥ ∈ f) then (f, r) else tryRules (f, r)

tryRules (f, r) =
  case r of
    (x : y) → let (f', r') = tryRules (f, y)
              in maybe (f', {x} ∪ r') (λc. tryRules c) (tryRule x (f, r))
    ∅       → (f, r)

tryRule x (f, r) =
```

$$\begin{aligned}&\text{let } h = \text{head } x\\&\quad\ b = \text{body } x\\&\text{in if } (\forall a \in b.a \in f) \text{ then } \mathit{Just}\ (\{h\} \cup f, r \setminus \{x\}) \text{ else } \mathit{Nothing}\end{aligned}$$

The functions *facts* and *rules* obtain the facts and the rules, respectively, from the constraint set C. The function *head* returns the atomic value that is in the head position of a clause and the function *body* returns the set of all literals that occur in the body of a clause.

6.2 Interpreting the results

The solving algorithm from the previous section is a generally applicable procedure for determining the consistency of a set of Horn constraints. What does the result of this algorithm tell us about the strictness behaviour of a function? In the initialization phase constraint sets were generated to determine the strictness of a function in each of its arguments. In each set a constraint was added that initialized the strictness value of the result to $\bot$.

By solving such a constraint set, we have verified whether initializing an argument of a function to the bottom value of its strictness domain and initializing all additional arguments to the top values of their strictness domains is consistent with initializing the strictness value of the function result to $\bot$. As we have seen in Chapter 3, if this is indeed consistent then the function is strict in the argument of which the strictness value was initialized to the bottom value of its strictness domain.

6.3 Intermediate solving

Thus far we have assumed that we are only interested in the strictness information of an entire expression. But with a small extension we can obtain the strictness information of all let definitions that occur in such an expression. We observe that all constraints that are needed to check the strictness of a let-defined expression appear in the constraint set before the body of the let expression in which the expression is defined. We introduce a new form of constraint, the *solve constraint*, that tells the solver to solve all constraints up to this solve constraint in the constraint set. A solve constraint is of the form

$$\mathit{Solve}\ x\ e\ \tau$$

Here x is the name of the let-defined expression and e is its definition. This information is needed so we can show it together with the derived strictness information. The strictness type τ of the let-defined expression is needed for the initialization phase of the intermediate solving.

The solve constraint is generated by extending the rule for let expressions

($\text{Let}^{\text{BU}}_{strict}$) as follows

$$(\text{Let}'^{\text{BU}}_{strict})\frac{\begin{array}{c} M, \mathcal{A}_1, e_1 :: t_1 \vdash^{\text{BU}}_{strict} (C_1, \tau_1) \quad M, \mathcal{A}_2, e_2 :: t_2 \vdash^{\text{BU}}_{strict} (C_2, \tau_2) \quad t \vdash \tau \\ C = C_1 \cup \{\tau_2 \leq_s \tau\} \cup (\mathcal{A}_1 \leq_M \{x : \tau_1\}) \cup \\ \{\mathit{Solve}\ x\ e_1\ \tau_1\} \cup (\mathcal{A}_2 \leq_M \{x : \tau_1\}) \cup C_2 \end{array}}{M, (\mathcal{A}_1 \cup \mathcal{A}_2) \backslash x, (\texttt{let}\ x = (e_1 :: t_1)\ \texttt{in}\ (e_2 :: t_2)) :: t \vdash^{\text{BU}}_{strict} (C, \tau)}$$

Note that the solve constraint is added to the constraint set directly after the constraints for the let definition, thus marking the end of the subset of constraints that are relevant for the let-defined expression. The solver must respect this ordering of constraints to find the correct strictness information for the let-defined expression.

We then use the following strategy to solve the constraint set: directly after inference (thus before instantiation of the instance constraints) the constraint set C is traversed in order. Two scenarios are possible:

1. During the traversal a solve constraint s is encountered. The subset C' of constraints that appear in C before s must be solved. This means that all instance constraints in C' are instantiated first. Then the strictness variables that appear in the strictness type bound by s are initialized. A constraint set that is obtained this way is then checked for consistency by the Horn solver. The derived strictness information is stored together with the corresponding name and definition, so that this information can be shown to the user later on. The solve constraint s is then removed from the original constraint set C (one might say that the solve constraint itself has been solved and can therefore be discarded). This set is again traversed until either another solve constraint is encountered, in which case the same procedure takes place, or

2. the constraint set has been traversed completely. In this case all solve constraints have been dealt with (i.e., we have derived strictness information for all intermediate let definitions) and we must now derive the strictness behaviour of the expression as a whole. This is done in the usual way: first all instance constraints are instantiated, then the strictness variables that occur in the strictness type derived by the inference algorithm are initialized, and finally the resulting constraint set is checked for consistency by the Horn solver.

If explicit strictness typing is allowed, the strictness annotation of a let-defined expression (if supplied) must also be stored in the solve constraint when it is generated. In the first scenario we then have to compare this strictness annotation with the information obtained by solving the constraint set C'.

Conclusion

The constraints that remain to be solved after inference and instantiation can be straightforwardly rewritten to Horn. However, these constraints only describe

the relation between the various strictness variables; some of the strictness variables need to be initialized in order to determine the strictness of the underlying function in its arguments. For every argument the original constraint set is duplicated and extended with the appropriate constraints to determine the strictness of the function in this argument. Such a constraint set can then be fed to an algorithm that checks the consistency of Horn constraint sets. If the set turns out to be consistent, the function is strict in the examined argument. By introducing solve constraints, we can solve a subset of the constraint set that is obtained by the inference algorithm. This is useful when we are interested in the strictness behaviour of intermediate let definitions.

Chapter 7

Implementation and test results

Combining the inference algorithm of Chapter 5 and the Horn solver of Chapter 6 we have built a standalone strictness analysis in Haskell that handles the subset of Helium as described in Chapter 5 and can additionally deal with strictness annotations. This implementation can be obtained from the Top project page [13].

We used a monadic approach to keep track of the state of the inference process. The collected constraints, monomorphic strictness type variables and intermediate results are stored in this state. For the representation of strictness types we have reused the data type representation of regular types from Top, which proved to be general enough for this purpose. Consequently, substitution, unification, and similar operations on strictness types came 'for free'.

Horn constraints are currently not explicitly supported by Top, so we used our own representation for constraints and strictness schemes that can describe these constraints. Unfortunately, we could therefore not reuse the generalization and instantiation operations supplied by Toand had to define similar operations ourselves, which increased the amount of code significantly (we roughly estimate the extra amount of code to be 25% of the total number of lines of code). Representation of Horn constraints is therefore in our opinion a useful addition to the Top framework; it is then possible to represent strictness schemes and all forms of constraints as data types that are defined in Top. We can thus take advantage of Top's facilities for manipulation of constraints and type schemes. As a result, we can focus more on the implementation of the inference rules and the solver.

The Horn solver has been built as an independent Haskell module that can be reused for other applications. Since it uses a different, more general, representation of Horn clauses, the strictness inference system contains a facility to rewrite constraints to this representation.

We have tested our implementation by comparing its output and performance for a number of example programs with GHC's strictness analysis. In most cases both strictness analyses found the same strictness information for a program. In some cases GHC gives more specific information. For instance, for

the example program

```
zero = \x -> 0
```

GHC finds that the function `zero` is *certainly not strict* in its argument, since it is not used in the body of the lambda abstraction. In this kind of situation, our strictness analysis finds the value $\top$ for such an argument.

For a program like

```
g = \x y -> let f = \b y z -> if b then y else z
            in  f True x y
```

GHC finds that, since the guard expression of the conditional expression evaluates to `True` for every invocation of `f`, the then-branch will always be evaluated. Thus `g` is strict in its first argument. Since our strictness inferencer does not use this kind of dataflow information, it has to assume that there is a possibility that the else-branch is evaluated when `f` is invoked. Therefore, it will find that it is unknown whether `g` is strict in any of its arguments.

Probably GHC gives better results in the situations described above, because other optimizations and analyses have taken place before strictness analysis is performed by the compiler. After inlining and partial evaluation the function `g` above might have been reduced to

```
g = \x y -> x
```

Clearly, a strictness analysis performed on this reduced definition will find that it is strict in its first argument. However, as we have mentioned before, we want to avoid rewriting of the original program as much as possible. For the function `zero` a simple syntactic check has probably found out that `x` is not used in the body of the lambda abstraction.

For small programs our strictness analysis runs relatively fast. However, the nesting of let expressions slows down the analysis significantly. It seems that instantiation is an expensive operation, so it would be interesting to look into the possibilities for optimizatition of this phase.

Chapter 8

Conclusion and future work

Thus far type inference was the only program analysis implemented within the Top framework. A program analysis implemented within this framework should be customizable and should provide useful feedback to the user. To achieve this, the underlying algorithm should collect the desired information, in the form of constraints, in a bottom-up traversal through the abstract syntax tree of a program. We have implemented strictness analysis that has the desired properties and should fit well in this framework.

The goal of strictness analysis is to determine which arguments of a function can be evaluated eagerly, rather than lazily, without changing the termination behaviour of the function. The traditional technique for strictness analyis is abstract interpretation, where a concrete value is replaced by an abstract value representing its strictness. Recent work by Kuo and Mishra [7] and Glynn *et al.* [3] shows how to perform strictness analysis as a special form of constraint-based type inference, representing the strictness of a value as a constrained strictness type. Although this approach comes close to how we want to implement strictness analysis in the Top framework, it is not directly usable to us since the inference algorithms are top-down oriented and the underlying language differs from Helium at some relevant points.

In this thesis we have introduced a strictness inference algorithm that largely overcomes these problems. Concepts of both Top's type inference algorithm and Glynn *et al.*'s strictness inference algorithm have been reused in formulating this algorithm. After inference the generated instance constraints are solved. In the next phase of the strictness analysis the remaining constraints are rewritten to Horn constraints that can be solved by a simple algorithm. Before the contraint set is solved, first some constraints are added to initialize the strictness variables that correspond to the arguments to the function that is examined.

Currently, a relevant subset of Helium is supported. Notable omissions are data types and let expressions with multiple definitions. Furthermore patterns other than variables are not supported. Dealing with these shortcomings remains future work. To take advantage of explicit strictness typing, we would like to extend the syntax of Helium with strictness annotations. The strictness analysis remains to be integrated in the Helium compiler, such that the compiler can use strictness information for code optimization.

Furthermore, the current implementation of our strictness analysis is relatively slow, especially compared to the performance of GHC. We might achieve better performance by optimizing the implementation of the strictness inference algorithm, especially the instantiation phase, and by using a faster Horn solver, written in C for example, to solve the constraints.

Integration of Horn constraints in the Top framework might result in a more compact, and probably more efficient, implementation. We now reuse three modules from Top (for types, substitutions and unifications), against three self-written modules (one containing basic definitions, one containing the inference algorithm and the instantiation operations, and one containing the Horn solver). We think that we can reuse at least three additional Top modules (for the representation and manipulation of constraints and type schemes) and need about 25% less self-written code when Horn constraints are integrated.

We would like to compare the performance and results of our strictness analysis with that of Glynn *et al.*'s strictness analysis. Unfortunately, no up to date implementation of their analysis was available at the time we tested our implementation. A comparison with the strictness analyser of the Clean compiler might also be useful, especially since it supports explicit strictness typing. Additionally, we believe that better results can be obtained if dataflow information is taken into account when strictness analysis is performed.

Bibliography

[1] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. The theory and practice of strictness for higher order functions. *Science of Computer Programming*, 7:249 – 278, 1986.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th POPL*, pages 238 – 252. ACM Press, January 1977.

[3] Kevin Glynn, Peter J. Stuckey, and Martin Sulzmann. Effective strictness analysis with HORN constraints. In P. Cousot, editor, *SAS 2001, LNCS*, volume 2126, pages 73 – 92. Springer-Verlag, January 2001.

[4] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Constraint based type inferencing in Helium. In M.-C. Silaghi and M. Zanker, editors, *Workshop Proceedings of Immediate Applications of Constraint Programming*, pages 59 – 80, Cork, September 2003.

[5] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.

[6] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.

[7] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: a new perspective based on type inference. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 260 – 272. ACM Press, 1989.

[8] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.

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

[10] Eric Nocker. Strictness analysis using abstract reduction. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 255 – 265. ACM Press, 1993.

[11] The Haskell Home Page. http://www.haskell.org.

[12] Rinus Plasmeijer and Mark van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[13] Top. http://www.cs.uu.nl/groups/ST/twiki/bin/view/Center/Top.

[14] Phil Wadler. Strictness analysis on non-flat domains. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 266 – 275. Ellis Horwood, 1987.