The Calculated Typer

ZAC GARBY, University of Nottingham, United Kingdom
PATRICK BAHR, IT University of Copenhagen, Denmark
GRAHAM HUTTON, University of Nottingham, United Kingdom

We present a calculational approach to the design of type checkers, showing how they can be derived from behavioural specifications using equational reasoning. In addition, we show how the calculations can be simplified by taking an algebraic approach based on fold fusion, and further improved by taking a constraint-based approach to solving and composing fusion preconditions. We illustrate our methodology with three examples of increasing complexity, starting with a simple expression language, then adding support for exceptions, and finally considering a version of the lambda calculus.

1 Introduction

Type checking is the process of verifying that programs are constructed in accordance with a set of typing rules [Pierce 2002]. For example, if we specify that addition requires two integer arguments and produces an integer result, then the expression $1 + 2$ is well-typed because both arguments are integers, whereas $1 + \mathit{True}$ is ill-typed because the second argument is not an integer. Moreover, we can say that the expression $1 + 2$ has integer type because it produces an integer result, while $1 + \mathit{True}$ has a type error due to its improper construction.

In this article, we focus on the problem of defining *type-checking functions* that analyse the structure of a program to determine if it is well-typed. Moreover, if a program is well-typed, the type-checking function returns the type of the program, and otherwise it returns a type error. Traditionally, such type checkers are defined by hand based on a pre-determined set of typing rules that specify valid ways to construct programs. Here we take a different approach, showing how to *calculate* type-checking functions directly from specifications of their behaviour.

The starting point for our calculational approach is a semantics for the language being considered, expressed as an evaluation function. We then formulate a specification that captures the desired behaviour of the type-checking function, which here means it returns the type of the value that would be produced if evaluation succeeds, or returns a type error if evaluation may fail due to the input being badly formed. Finally, we use equational reasoning techniques to calculate an implementation for the type-checking function that satisfies the specification.

The calculational approach to type checker design has two key benefits. First of all, the resulting type checkers are *correct-by-construction* [Backhouse 2003], eliminating the need for separate correctness proofs. And secondly, the approach provides a systematic way to *discover* typing rules for a language, and to explore alternative design choices during the calculation process.

We develop our approach in three stages. We begin with a first-principles approach using explicit recursive definitions and inductive reasoning. We then simplify the calculations by adopting an algebraic approach using a *fold* operator and its fusion property. Finally, we refine the methodology further by using a constraint-based approach to solving and composing fusion preconditions. To demonstrate the utility of our approach we present three examples of increasing complexity, starting with a simple expression language with conditionals, then adding support for exception handling, and finally considering a version of the lambda calculus.

All our programs and calculations are written in Haskell [Marlow et al. 2010], but the ideas are applicable in any functional language. Because the calculations are the central focus, they are

Authors' Contact Information: Zac Garby, University of Nottingham, Nottingham, United Kingdom, zac.garby@nottingham.ac.uk; Patrick Bahr, IT University of Copenhagen, Copenhagen, Denmark, paba@itu.dk; Graham Hutton, University of Nottingham, Nottingham, United Kingdom, graham.hutton@nottingham.ac.uk.

typically presented in detail rather than being compressed or omitted. Haskell code for each of the examples is freely available online as supplementary material.

2 Positivity Checking

Type checking is an example of the general idea of static analysis [Nielson et al. 1999], which seeks to reason about the behaviour of programs without executing them. Prior to considering type checking, we first consider a simpler example, to illustrate how a static analysis can be calculated from a specification of its behaviour using equational reasoning techniques.

Consider a simple type of arithmetic expressions built up from integer values using an addition operator, together with a semantics that evaluates an expression to an integer:

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

$$\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \rightarrow \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ n) \quad\ \ = n\\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}$$

Suppose now that we wish to define a function $\mathit{isPos} :: \mathit{Expr} \rightarrow \mathit{Bool}$ that decides if an expression is positive, without evaluating the expression. The desired behaviour is as follows:

$$\mathit{isPos}\ e \quad\Rightarrow\quad \mathit{eval}\ e > 0$$

That is, if an expression is classified as positive, then evaluation should give a positive result. Note that the specification is an implication rather than an equivalence, because in general it may not be possible to decide if an expression is positive without evaluating it.

To calculate a definition for *isPos*, we proceed by induction on the expression *e*. In each case, we start with the right-hand side of the specification, $\mathit{eval}\ e > 0$, and seek to strengthen it to something of the form $\mathit{isPos}\ e$ by defining a suitable clause for the function in this case.

Case: *Val n*

$$\begin{aligned}
&\quad \mathit{eval}\ (\mathit{Val}\ n) > 0\\
\Leftrightarrow&\quad \{\text{ applying } \mathit{eval}\ \}\\
&\quad n > 0\\
\Leftrightarrow&\quad \{\text{ define: } \mathit{isPos}\ (\mathit{Val}\ n) = n > 0\ \}\\
&\quad \mathit{isPos}\ (\mathit{Val}\ n)
\end{aligned}$$

Case: *Add x y*

$$\begin{aligned}
&\quad \mathit{eval}\ (\mathit{Add}\ x\ y) > 0\\
\Leftrightarrow&\quad \{\text{ applying } \mathit{eval}\ \}\\
&\quad \mathit{eval}\ x + \mathit{eval}\ y > 0\\
\Leftarrow&\quad \{\text{ addition preserves positivity }\}\\
&\quad \mathit{eval}\ x > 0 \wedge \mathit{eval}\ y > 0\\
\Leftarrow&\quad \{\text{ induction hypotheses }\}\\
&\quad \mathit{isPos}\ x \wedge \mathit{isPos}\ y\\
\Leftrightarrow&\quad \{\text{ define: } \mathit{isPos}\ (\mathit{Add}\ x\ y) = \mathit{isPos}\ x \wedge \mathit{isPos}\ y\ \}\\
&\quad \mathit{isPos}\ (\mathit{Add}\ x\ y)
\end{aligned}$$

The key step in the calculation uses the fact that addition preserves positivity, i.e. $n > 0$ and $m > 0$ implies $n + m > 0$, to transform the term being manipulated into a form to which the induction hypotheses can be applied. In conclusion, we have calculated the following definition:

```
isPos :: Expr → Bool
isPos (Val n)     = n > 0
isPos (Add x y) = isPos x ∧ isPos y
```

The approach that we will use to calculate a function determining the type of an expression is similar to the above, but in a more sophisticated setting.

3 Conditional Expressions

We now consider an expression language built up from basic values using addition and conditional operations, where a value is either an integer, a logical value, or an error value:

```
data Expr  = Val Value | Add Expr Expr | If Expr Expr Expr
data Value = I Int | B Bool | Error
```

The semantics is defined using a function that evaluates an expression, with the error value being returned if evaluation fails due to the expression being badly formed:

```
eval :: Expr → Value
eval (Val v)     = v
eval (Add x y) = add (eval x) (eval y)
eval (If x y z) = cond (eval x) (eval y) (eval z)
```

The operations *add* and *cond* on values formalise that addition requires two integers to succeed, while conditionals require a logical value to make a choice between two alternatives:

```
add :: Value → Value → Value
add (I n) (I m) = I (n + m)
add _     _     = Error

cond :: Value → Value → Value → Value
cond (B b) v w = if b then v else w
cond _     _ _ = Error
```

As we shall see, separating out these operations as named functions plays an important role in our development, as it allows us to exploit properties of these functions.

We could also consider other approaches to handling badly formed expressions in the semantics, such as replacing the explicit error value by the *Maybe* monad and defining an evaluation function *eval* :: *Expr* → *Maybe Value*, or using a big-step operational approach and defining an evaluation relation between expressions and values, which doesn't require an error value as relations can be partial. Here we choose to focus on the simple functional approach with an explicit error value, as this mirrors the first-principles approach used by Bahr and Hutton [2015].

4 Type Checking

Evaluation fails if an expression is badly formed. In this case, badly formed means that it contains a *type error*, such as attempting to add two values that are not integers. To formalise this idea, we first define a simple language of types, comprising integers, logical values and an error type, together with a function that abstracts from the contents of a value and returns its type:

```
data Type = INT | BOOL | ERROR

tval :: Value → Type
tval (I _)  = INT
tval (B _) = BOOL
tval Error = ERROR
```

Suppose now that we wish to define a function *texp* :: *Expr* → *Type* that returns the type of an expression, with the error type being returned if the expression is badly formed. How can we specify the desired behaviour of this function? A first attempt might be as follows:

$$texp\ e \quad = \quad tval\ (eval\ e)$$

This equation states that type checking an expression should give the same result as evaluating the expression and taking the type of the resulting value. While this formulation captures the basic idea, it is also too strong. To see why, consider the expression **if** *True* **then** 1 **else** *False*, here written in Haskell syntax. Evaluating this expression gives the value 1, which is an integer. However, for the type checker to determine this, it would need to use the condition's value to decide which branch applies. We do not want the type checker to observe values in this way, as in general this involves evaluation, which is precisely what the type checker aims to abstract away.

The above specification would therefore be unsatisfiable under this constraint. The root cause is that type checking is necessarily *approximate* rather than precise. We can formalise this idea by defining an ordering relation on types via the following class declaration:

```
instance Ord Type where
  (⩽) :: Type → Type → Bool
  t ⩽ t′ = t == ERROR ∨ t == t′
```

That is, *ERROR* is defined to be less than any type, while any two other types are related by the ordering if they are equal. As a Hasse diagram, the ordering can be illustrated as follows:

$$\begin{tikzcd}
INT \arrow[dr, dash] & & BOOL \arrow[dl, dash] \\
& ERROR &
\end{tikzcd}$$

We can view this as an *information* ordering, where $t \leqslant t'$ means that the type t contains less information than the type t'. This is the reverse of the usual subtype ordering, and is used to capture the approximate nature of type checking. We will sometimes find it convenient to use the opposite ordering $\geqslant$, which is defined in the usual way by $t \geqslant t'$ iff $t' \leqslant t$.

Note that the *Ord* class normally requires a total ordering, but for our purposes we only assume the $\leqslant$ ordering is partial and do not use properties or operations that depend on totality. Using this ordering, the behaviour of type checking can then be captured as follows:

$$texp\ e \quad \leqslant \quad tval\ (eval\ e)$$

That is, type checking either returns a type error, or gives the same result as evaluating the given expression and then taking the type of the resulting value.

Note that the above specification gives flexibility in how type checking is implemented, as there may be many possible definitions that satisfy the inequation, with some being more informative than others. Indeed, the trivial definition *texp e* = *ERROR* that always gives a type error is perfectly valid. The calculational methodology we develop naturally avoids such trivial solutions, and provides a systematic means for designing type checkers that satisfy the above specification.

5 Calculating the Type Checker

Rather than first defining the type checking function and then proving it is correct, we can also use the specification $texp\ e \leqslant tval\ (eval\ e)$ as the basis for *calculating* the definition of the function *texp*.

The calculation proceeds by induction on the expression e. For each case of e, we start with the right-hand side of the specification, $\mathit{tval}\ (\mathit{eval}\ e)$, and seek to strengthen it into something of the form $\mathit{texp}\ e$ by defining a suitable clause for the function in this case.

Case: $\mathit{Val}\ v$

$$\begin{array}{cl}
 & \mathit{tval}\ (\mathit{eval}\ (\mathit{Val}\ v)) \\
= & \{\ \text{applying } \mathit{eval}\ \} \\
 & \mathit{tval}\ v \\
= & \{\ \text{define: } \mathit{texp}\ (\mathit{Val}\ v) = \mathit{tval}\ v\ \} \\
 & \mathit{texp}\ (\mathit{Val}\ v)
\end{array}$$

Case: $\mathit{Add}\ x\ y$

$$\begin{array}{cl}
 & \mathit{tval}\ (\mathit{eval}\ (\mathit{Add}\ x\ y)) \\
= & \{\ \text{applying } \mathit{eval}\ \} \\
 & \mathit{tval}\ (\mathit{add}\ (\mathit{eval}\ x)\ (\mathit{eval}\ y))
\end{array}$$

Now we appear to be stuck, as no further definitions can be applied. However, as we are performing an inductive calculation, we can use the induction hypotheses for x and y, namely:

$$\begin{array}{rcl}
\mathit{texp}\ x & \leqslant & \mathit{tval}\ (\mathit{eval}\ x) \\
\mathit{texp}\ y & \leqslant & \mathit{tval}\ (\mathit{eval}\ y)
\end{array}$$

To use these hypotheses, it is clear that we need to distribute tval over the operation add on values in the term $\mathit{tval}\ (\mathit{add}\ (\mathit{eval}\ x)\ (\mathit{eval}\ y))$ to give a term of the form $\mathit{add}'\ (\mathit{tval}\ (\mathit{eval}\ x))\ (\mathit{tval}\ (\mathit{eval}\ y))$, for some operation add' on types. That is, we need to solve the inequation:

$$\mathit{tval}\ (\mathit{add}\ (\mathit{eval}\ x)\ (\mathit{eval}\ y)) \quad \geqslant \quad \mathit{add}'\ (\mathit{tval}\ (\mathit{eval}\ x))\ (\mathit{tval}\ (\mathit{eval}\ y))$$

First of all, we generalise from the specific values $\mathit{eval}\ x$ and $\mathit{eval}\ y$ to arbitrary values:

$$\mathit{tval}\ (\mathit{add}\ v\ w) \quad \geqslant \quad \mathit{add}'\ (\mathit{tval}\ v)\ (\mathit{tval}\ w)$$

Assuming this property then allows us to apply induction and complete the calculation:

$$\begin{array}{cl}
 & \mathit{tval}\ (\mathit{add}\ (\mathit{eval}\ x)\ (\mathit{eval}\ y)) \\
\geqslant & \{\ \text{assume: } \mathit{tval}\ (\mathit{add}\ v\ w) \geqslant \mathit{add}'\ (\mathit{tval}\ v)\ (\mathit{tval}\ w)\ \} \\
 & \mathit{add}'\ (\mathit{tval}\ (\mathit{eval}\ x))\ (\mathit{tval}\ (\mathit{eval}\ y)) \\
\geqslant & \{\ \text{induction hypotheses, assume: } \mathit{add}' \text{ is monotonic}\ \} \\
 & \mathit{add}'\ (\mathit{texp}\ x)\ (\mathit{texp}\ y) \\
= & \{\ \text{define: } \mathit{texp}\ (\mathit{Add}\ x\ y) = \mathit{add}'\ (\mathit{texp}\ x)\ (\mathit{texp}\ y)\ \} \\
 & \mathit{texp}\ (\mathit{Add}\ x\ y)
\end{array}$$

The induction step also requires that add' is monotonic, i.e. preserves the ordering on types:

$$\frac{t \leqslant t' \qquad u \leqslant u'}{\mathit{add}'\ t\ u \ \leqslant\ \mathit{add}'\ t'\ u'}$$

To discharge the assumptions made in the above calculation, it remains to define the operation $\mathit{add}' :: \mathit{Type} \longrightarrow \mathit{Type} \longrightarrow \mathit{Type}$, and show that it satisfies the required distributivity and monotonicity properties. In fact, we can calculate the definition for add' directly from its distributivity property by starting with the left-hand side of this property, $\mathit{tval}\ (\mathit{add}\ v\ w)$, and aiming to transform it into the form of the right-hand side, $\mathit{add}'\ (\mathit{tval}\ v)\ (\mathit{tval}\ w)$:

$$\begin{array}{ll}
& \mathit{tval}\ (\mathit{add}\ v\ w) \\
= & \{\ \text{applying } \mathit{add}\ \} \\
& \mathit{tval}\ (\textbf{case}\ (v, w)\ \textbf{of} \\
& \qquad (I\ n, I\ m) \quad \rightarrow I\ (n + m) \\
& \qquad (\_, \_) \qquad\quad \rightarrow \mathit{Error}) \\
= & \{\ \text{distribution, } \mathit{tval} \text{ is strict}\ \} \\
& \textbf{case}\ (v, w)\ \textbf{of} \\
& \qquad (I\ n, I\ m) \quad \rightarrow \mathit{tval}\ (I\ (n + m)) \\
& \qquad (\_, \_) \qquad\quad \rightarrow \mathit{tval}\ \mathit{Error}) \\
= & \{\ \text{applying } \mathit{tval}\ \} \\
& \textbf{case}\ (v, w)\ \textbf{of} \\
& \qquad (I\ n, I\ m) \quad \rightarrow \mathit{INT} \\
& \qquad (\_, \_) \qquad\quad \rightarrow \mathit{ERROR} \\
= & \{\ \text{unapplying } \mathit{tval}\ \} \\
& \textbf{case}\ (\mathit{tval}\ v, \mathit{tval}\ w)\ \textbf{of} \\
& \qquad (\mathit{INT}, \mathit{INT}) \rightarrow \mathit{INT} \\
& \qquad (\_, \_) \qquad\quad \rightarrow \mathit{ERROR} \\
= & \{\ \text{define: } \mathit{add}'\ \mathit{INT}\ \mathit{INT} = \mathit{INT};\ \mathit{add}'\ \_\ \_ = \mathit{ERROR}\ \} \\
& \mathit{add}'\ (\mathit{tval}\ v)\ (\mathit{tval}\ w)
\end{array}$$

In summary, we have been able to establish an equality between the two sides, which is stronger than the inequality that was required, and have therefore calculated the following definition:

$$\begin{array}{l}
\mathit{add}' :: \mathit{Type} \rightarrow \mathit{Type} \rightarrow \mathit{Type} \\
\mathit{add}'\ \mathit{INT}\ \mathit{INT} = \mathit{INT} \\
\mathit{add}'\ \_ \quad\ \ \_ \qquad = \mathit{ERROR}
\end{array}$$

That is, adding two integers gives an integer, while adding anything else gives a type error. We can show that *add′* is monotonic by considering monotonicity in each argument separately. For example, we can show monotonicity in the first argument as follows:

$$\begin{array}{ll}
& \mathit{add}'\ t\ u \leqslant \mathit{add}'\ t'\ u \\
\Leftrightarrow & \{\ \text{applying } \leqslant\ \} \\
& \mathit{add}'\ t\ u == \mathit{ERROR} \lor \mathit{add}'\ t\ u == \mathit{add}'\ t'\ u \\
\Leftarrow & \{\ \text{unapplying } \mathit{add}'\ \} \\
& t == \mathit{ERROR} \lor \mathit{add}'\ t\ u == \mathit{add}'\ t'\ u \\
\Leftarrow & \{\ \text{extensionality}\ \} \\
& t == \mathit{ERROR} \lor t == t' \\
\Leftrightarrow & \{\ \text{unapplying } \leqslant\ \} \\
& t \leqslant t'
\end{array}$$

The key step here is the second one, where we use the fact that *add′* returns an error if its first argument is an error. Monotonicity in the second argument follows similarly, because the definition for operation *add′* is symmetric in its two arguments.

Case: *Cond x y z*

This case proceeds in a similar manner to the case for addition, by assuming a distributivity property that allows the induction hypotheses to be applied:

$$
\begin{array}{ll}
& \mathit{tval}\ (\mathit{eval}\ (\mathit{Cond}\ x\ y\ z)) \\
= & \{\ \text{applying } \mathit{eval}\ \} \\
& \mathit{tval}\ (\mathit{cond}\ (\mathit{eval}\ x)\ (\mathit{eval}\ y)\ (\mathit{eval}\ z)) \\
\geqslant & \{\ \text{assume: } \mathit{tval}\ (\mathit{cond}\ c\ v\ w) \geqslant \mathit{cond'}\ (\mathit{tval}\ c)\ (\mathit{tval}\ v)\ (\mathit{tval}\ w)\ \} \\
& \mathit{cond'}\ (\mathit{tval}\ (\mathit{eval}\ x))\ (\mathit{tval}\ (\mathit{eval}\ y))\ (\mathit{tval}\ (\mathit{eval}\ z)) \\
\geqslant & \{\ \text{induction hypotheses, assume: } \mathit{cond'} \text{ is monotonic}\ \} \\
& \mathit{cond'}\ (\mathit{texp}\ x)\ (\mathit{texp}\ y)\ (\mathit{texp}\ z) \\
= & \{\ \text{define: } \mathit{texp}\ (\mathit{Cond}\ x\ y\ z) = \mathit{cond'}\ (\mathit{texp}\ x)\ (\mathit{texp}\ y)\ (\mathit{texp}\ z)\ \} \\
& \mathit{texp}\ (\mathit{Cond}\ x\ y\ z)
\end{array}
$$

We can then calculate the definition for *cond′* from its distributivity property by starting with the left-hand side, and aiming for the right-hand side:

$$
\begin{array}{ll}
& \mathit{tval}\ (\mathit{cond}\ c\ v\ w) \\
= & \{\ \text{applying } \mathit{cond}\ \} \\
& \mathit{tval}\ (\textbf{case}\ c\ \textbf{of} \\
& \quad B\ b \longrightarrow \textbf{if}\ b\ \textbf{then}\ v\ \textbf{else}\ w \\
& \quad \_ \quad \longrightarrow \mathit{Error}) \\
= & \{\ \text{distribution, } \mathit{tval} \text{ is strict}\ \} \\
& \textbf{case}\ c\ \textbf{of} \\
& \quad B\ b \longrightarrow \textbf{if}\ b\ \textbf{then}\ \mathit{tval}\ v\ \textbf{else}\ \mathit{tval}\ w \\
& \quad \_ \quad \longrightarrow \mathit{tval}\ \mathit{Error} \\
= & \{\ \text{applying } \mathit{tval}\ \} \\
& \textbf{case}\ c\ \textbf{of} \\
& \quad B\ b \longrightarrow \textbf{if}\ b\ \textbf{then}\ \mathit{tval}\ v\ \textbf{else}\ \mathit{tval}\ w \\
& \quad \_ \quad \longrightarrow \mathit{ERROR} \\
\geqslant & \{\ \text{lemma below, } \textbf{case} \text{ is monotonic}\ \} \\
& \textbf{case}\ c\ \textbf{of} \\
& \quad B\ b \longrightarrow \textbf{if}\ \mathit{tval}\ v == \mathit{tval}\ w\ \textbf{then}\ \mathit{tval}\ v\ \textbf{else}\ \mathit{ERROR} \\
& \quad \_ \quad \longrightarrow \mathit{ERROR} \\
= & \{\ \text{unapplying } \mathit{tval}\ \} \\
& \textbf{case}\ (\mathit{tval}\ c)\ \textbf{of} \\
& \quad \mathit{BOOL} \longrightarrow \textbf{if}\ \mathit{tval}\ v == \mathit{tval}\ w\ \textbf{then}\ \mathit{tval}\ v\ \textbf{else}\ \mathit{ERROR} \\
& \quad \_ \quad \longrightarrow \quad \mathit{ERROR} \\
= & \{\ \text{define: } \mathit{cond'}\ \mathit{BOOL}\ t\ t' = \textbf{if}\ t == t'\ \textbf{then}\ t\ \textbf{else}\ \mathit{ERROR};\ \mathit{cond'}\ \_\ \_\ \_ = \mathit{ERROR}\ \} \\
& \mathit{cond'}\ (\mathit{tval}\ c)\ (\mathit{tval}\ v)\ (\mathit{tval}\ w)
\end{array}
$$

The key step here is the following lemma, which allows us to replace a conditional that depends on a logical value by a conditional that only depends on the types in the branches:

$$\textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ t' \quad \geqslant \quad \textbf{if}\ t == t'\ \textbf{then}\ t\ \textbf{else}\ \mathit{ERROR}$$

We can prove this lemma by case analysis on the logical value $t == t'$. If this value is true, the lemma simplifies to $(\textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ t) \geqslant t$, which in turn simplifies to $t \geqslant t$ assuming b is well-defined, which is then true by reflexivity. If $t == t'$ is false, the lemma simplifies to $(\textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ t') \geqslant \mathit{ERROR}$, which is true because *ERROR* is the smallest type by definition.

In summary, we have calculated the following definition for the operation *cond′* on types, which formalises the idea that a conditional expression requires a logical value and two branches that have the same type, otherwise it gives a type error:

```
cond′ :: Type → Type → Type → Type
cond′ BOOL t  t′ = if t == t′ then t else ERROR
cond′ _      _ _ = ERROR
```

Showing that *cond′* is monotonic on types proceeds in a similar manner to the function *add′*, using the fact that *cond′* preserves type errors in all arguments.

Putting everything together, we have calculated the following definition for the type checking function *texp* in terms of newly defined operations *add′* and *cond′* on types:

```
texp :: Expr → Type
texp (Val v)     = tval v
texp (Add x y) = add′ (texp x) (texp y)
texp (If x y z) = cond′ (texp x) (texp y) (texp z)

add′ :: Type → Type → Type
add′ INT INT  = INT
add′ _    _    = ERROR

cond′ :: Type → Type → Type → Type
cond′ BOOL t  t′ = if t == t′ then t else ERROR
cond′ _      _ _ = ERROR
```

6 Algebraic Approach

An important aspect of our development above was the use of auxiliary operations *add* and *cond* on values, and the introduction of operations *add′* and *cond′* on types. In particular, separating these out as named functions allowed us to state and exploit algebraic properties of these operations, namely distributivity and monotonicity, to simplify and guide our calculations.

However, we can take the algebraic approach further, and there are benefits from doing so. The starting point is to define a *fold* operator for expressions [Meijer et al. 1991], which takes the operation to apply for each form of expression as a parameter in the definition:

```
folde :: (Value → a) → (a → a → a) → (a → a → a → a) → Expr → a
folde val add cond = f
  where
    f (Val v)     = val v
    f (Add x y) = add (f x) (f y)
    f (If x y z) = cond (f x) (f y) (f z)
```

Using this operator, we can then redefine evaluation and type checking in a more concise manner by simply supplying the appropriate operations for values, addition and conditionals:

```
eval :: Expr → Value
eval = folde id add cond

texp :: Expr → Type
texp = folde tval add′ cond′
```

The fold operator also has a useful *fusion* property, which allows a function h that satisfies suitable distributivity, or *homomorphism* properties to be fused together with a fold using one collection of operations, *val*, *add* and *cond*, to give a fold over another collection, *val′*, *add′* and *cond′*. In our setting, we use the following variant of fusion where equality = is replaced by a pre-order $\geqslant$, under which the operations *add′* and *cond′* are required to be monotonic:

$$\frac{\begin{array}{rcl} h\ (val\ x) & \geqslant & val'\ x \\ h\ (add\ x\ y) & \geqslant & add'\ (h\ x)\ (h\ y) \\ h\ (cond\ x\ y\ z) & \geqslant & cond'\ (h\ x)\ (h\ y)\ (h\ z) \end{array}}{h\ (folde\ val\ add\ cond\ e)\ \geqslant\ folde\ val'\ add'\ cond'\ e}$$

This fusion property can be proved by straightforward induction on expressions. It now becomes evident that the correctness of type checking is just an application of fusion. In particular, the specification $tval\ (eval\ e) \geqslant texp\ e$ can now be expanded to:

$$tval\ (folde\ id\ add\ cond\ e) \ \geqslant\ folde\ tval\ add'\ cond'\ e$$

Hence, by fusion, this property is true if the operations *add′* and *cond′* are monotonic, and the function *tval* satisfies the following homomorphism properties:

$$\begin{array}{rcl} tval\ (id\ x) & \geqslant & tval\ x \\ tval\ (add\ x\ y) & \geqslant & add'\ (tval\ x)\ (tval\ y) \\ tval\ (cond\ x\ y\ z) & \geqslant & cond'\ (tval\ x)\ (tval\ y)\ (tval\ z) \end{array}$$

The first property is trivially true by reflexivity, while the second and third lead to the same calculations and resulting definitions for *add′* and *cond′* as shown in the previous section.

In conclusion, using an algebraic approach allows the type checking function $texp :: Expr \rightarrow Type$ to be obtained in a more principled manner than previously. In particular, using the the fold operation and its associated fusion property makes explicit the key property required for the type checking function to be correct, namely that the function $tval :: Value \rightarrow Type$ is a homomorphism between basic operations on values (*add* and *cond*) and those on types (*add′* and *cond′*).

7 Constraint Approach

As shown in the previous section, the correctness of type checking depends on certain homomorphism properties. For example, for conditionals we must establish that:

$$tval\ (cond\ x\ y\ z) \ \geqslant\ cond'\ (tval\ x)\ (tval\ y)\ (tval\ z)$$

Earlier we showed how this property can be used as the basis for calculating the definition for the operation *cond′* on types. However, an unsatisfactory aspect of the calculation was that it required ‘inventing’ the key lemma that makes the calculation work:

$$\mathbf{if}\ b\ \mathbf{then}\ t\ \mathbf{else}\ t' \ \geqslant\ \mathbf{if}\ t == t'\ \mathbf{then}\ t\ \mathbf{else}\ ERROR$$

However, there is another approach to calculating the definition for *cond′* that is both simpler and avoids the need to invent a lemma. The approach is based on using the homomorphism property that specifies the desired behaviour of *cond′* to derive a number of simpler constraints, which can then be solved collectively to give a definition for *cond′* itself.

We begin by observing that the operation *cond* on values is defined by case analysis on its first argument. We then proceed by considering all four possible cases of this argument, and simplifying the homomorphism property for conditionals in each of these cases.

Case: $x = B\ True$

$$
\begin{array}{ll}
& tval\ (cond\ (B\ True)\ y\ z) \geqslant cond'\ (tval\ (B\ True))\ (tval\ y)\ (tval\ z) \\
\Leftrightarrow & \{\text{ applying } cond,\ tval\,\} \\
& tval\ y \geqslant cond'\ BOOL\ (tval\ y)\ (tval\ z) \\
\Leftarrow & \{\text{ generalising to arbitrary types }\} \\
& t \geqslant cond'\ BOOL\ t\ t'
\end{array}
$$

Case: $x = B\ False$

$$
\begin{array}{ll}
& tval\ (cond\ (B\ False)\ y\ z) \geqslant cond'\ (tval\ (B\ False))\ (tval\ y)\ (tval\ z) \\
\Leftrightarrow & \{\text{ applying } cond,\ tval\,\} \\
& tval\ z \geqslant cond'\ BOOL\ (tval\ y)\ (tval\ z) \\
\Leftarrow & \{\text{ generalising to arbitrary types }\} \\
& t' \geqslant cond'\ BOOL\ t\ t'
\end{array}
$$

Case: $x = I\ n$

$$
\begin{array}{ll}
& tval\ (cond\ (I\ n)\ y\ z) \geqslant cond'\ (tval\ (I\ n))\ (tval\ y)\ (tval\ z) \\
\Leftrightarrow & \{\text{ applying } cond,\ tval\,\} \\
& tval\ Error \geqslant cond'\ INT\ (tval\ y)\ (tval\ z) \\
\Leftrightarrow & \{\text{ applying } tval\,\} \\
& ERROR \geqslant cond'\ INT\ (tval\ y)\ (tval\ z) \\
\Leftarrow & \{\text{ generalising to arbitrary types }\} \\
& ERROR \geqslant cond'\ INT\ t\ t'
\end{array}
$$

Case: $x = Error$

$$
\begin{array}{ll}
& tval\ (cond\ Error\ y\ z) \geqslant cond'\ (tval\ Error)\ (tval\ y)\ (tval\ z) \\
\Leftrightarrow & \{\text{ applying } cond,\ tval\,\} \\
& tval\ Error \geqslant cond'\ ERROR\ (tval\ y)\ (tval\ z) \\
\Leftrightarrow & \{\text{ applying } tval\,\} \\
& ERROR \geqslant cond'\ ERROR\ (tval\ y)\ (tval\ z) \\
\Leftarrow & \{\text{ generalising to arbitrary types }\} \\
& ERROR \geqslant cond'\ ERROR\ t\ t'
\end{array}
$$

The above reasoning shows that the following four constraints for the operation *cond′* are together sufficient to satisfy the homomorphism property for conditionals:

$$
\begin{array}{rcl}
cond'\ BOOL\ t\ t' & \leqslant & t \\
cond'\ BOOL\ t\ t' & \leqslant & t' \\
cond'\ INT\ t\ t' & \leqslant & ERROR \\
cond'\ ERROR\ t\ t' & \leqslant & ERROR
\end{array}
$$

The last two constraints have a unique solution, given by defining $cond'\ INT\ t\ t' = ERROR$ and $cond'\ ERROR\ t\ t' = ERROR$, because *ERROR* is the smallest type. In turn, if we assume a greatest lower bound operator $\sqcap$ on types, then the first two constraints are together equivalent to

$$cond'\ BOOL\ t\ t' \ \leqslant\ t \sqcap t'$$

by the universal property of greatest lower bounds:

$$x \leqslant y \quad \text{and} \quad x \leqslant z \quad \text{iff} \quad x \leqslant y \sqcap z$$

Hence, we can define $cond'\ BOOL\ t\ t' = t \sqcap t'$ as the optimal, i.e. greatest or most informative, solution to the first two constraints. Putting all of the above reasoning together, we conclude that the *cond′* operation can be defined by the following three equations,

$$\begin{array}{l} cond'\ BOOL\ \ t\ t' = t \sqcap t' \\ cond'\ INT\ \ \ \ \ t\ t' = ERROR \\ cond'\ ERROR\ t\ t' = ERROR \end{array}$$

which can then be simplified by using the wildcard pattern to combine the last two cases:

$$\begin{array}{l} cond'\ BOOL\ t\ \ t' = t \sqcap t' \\ cond'\ \_\ \ \ \ \ \ \ \ \_\ \_ = ERROR \end{array}$$

For the ordering relation $\leqslant$ on types that we are using, where $t \leqslant t'$ iff $t = ERROR$ or $t = t'$, the greatest lower bound operator is defined simply as:

$$\begin{array}{l} (\sqcap) :: Type \rightarrow Type \rightarrow Type \\ t \sqcap t' = \textbf{if}\ t == t'\ \textbf{then}\ t\ \textbf{else}\ ERROR \end{array}$$

In summary, we have shown how the previous definition for the *cond′* operation can be calculated in a principled manner that does not require the invention of a lemma:

$$\begin{array}{l} cond' :: Type \rightarrow Type \rightarrow Type \rightarrow Type \\ cond'\ BOOL\ t\ \ t' = \textbf{if}\ t == t'\ \textbf{then}\ t\ \textbf{else}\ ERROR \\ cond'\ \_\ \ \ \ \ \ \ \ \_\ \_ = ERROR \end{array}$$

The same kind of constraint-based reasoning can also be used to calculate the operation *add′* on types from the homomorphism property that it must satisfy:

$$tval\ (add\ x\ y) \ \ \geqslant \ \ add'\ (tval\ x)\ (tval\ y)$$

The operation *add* on values is defined by case analysis on its two arguments, so we proceed by considering the two cases for the definition separately.

Case: $x = I\ n$ and $y = I\ m$

$$\begin{array}{ll} & tval\ (add\ (I\ n)\ (I\ m)) \geqslant add'\ (tval\ (I\ n))\ (tval\ (I\ m)) \\ \Leftrightarrow & \{\text{applying } add,\ tval\} \\ & tval\ (I\ (n + m)) \geqslant add'\ INT\ INT \\ \Leftrightarrow & \{\text{applying } tval\} \\ & INT \geqslant add'\ INT\ INT \end{array}$$

Case: $x \neq I\ n$ or $y \neq I\ m$

$$\begin{array}{ll} & tval\ (add\ x\ y) \geqslant add'\ (tval\ x)\ (tval\ y) \\ \Leftrightarrow & \{\text{applying } add\} \\ & tval\ Error \geqslant add'\ (tval\ x)\ (tval\ y) \\ \Leftrightarrow & \{\text{applying } tval\} \\ & ERROR \geqslant add'\ (tval\ x)\ (tval\ y) \\ \Leftarrow & \{\text{generalising to arbitrary types}\} \\ & ERROR \geqslant add\ t\ t',\ \ \text{if } t \neq INT \text{ or } t' \neq INT \end{array}$$

The above reasoning shows that the following two constraints for the operation *add′* are together sufficient to satisfy the homomorphism property for addition:

$$\begin{aligned} \mathit{add}'\ \mathit{INT}\ \mathit{INT} &\leqslant \mathit{INT} \\ \mathit{add}'\ t\ t' &\leqslant \mathit{ERROR} \qquad \text{if } t \neq \mathit{INT} \text{ or } t' \neq \mathit{INT} \end{aligned}$$

The optimal, i.e. greatest or most informative, solution to these inequations is to strengthen each to an equality by defining $\mathit{add}'\ \mathit{INT}\ \mathit{INT} = \mathit{INT}$ and $\mathit{add}'\ t\ t' = \mathit{ERROR}$ if $t \neq \mathit{INT}$ or $t' \neq \mathit{INT}$, which can then be simplified using the wildcard pattern to give the following definition:

```
add′ :: Type → Type → Type
add′ INT INT = INT
add′ _   _   = ERROR
```

In conclusion, adopting a constraint-based approach to solving the required homomorphism properties allows the basic type checking operations *add′* and *cond′* to be obtained in a simpler way than previously, and in a manner that does not require inventing a lemma.

8 Composing Constraints

In this section we improve the constraint-based approach from the previous section, by showing how type checking operations can be defined by directly composing their constraints. This approach also ensures that the resulting operations are monotonic by construction, and provides additional flexibility to handle more complex types in the next section.

We introduce and explain the compositional approach by considering the first constraint that we derived for the operation *cond′* on types in the previous section:

$$\mathit{cond}'\ \mathit{BOOL}\ t\ t' \leqslant t \tag{1}$$

We will transform this constraint into an equivalent form that can be used to compose the definition for *cond′* directly. We proceed in a series of steps. First of all, we observe that under the assumption that *cond′* is monotonic, the above constraint is equivalent to:

$$\mathit{cond}'\ s\ t\ t' \leqslant t \qquad \text{if } s \leqslant \mathit{BOOL} \tag{2}$$

That is, the explicit matching on *BOOL* in the original constraint is replaced by the side-condition that the type of the first argument is bounded above by *BOOL*. To verify that (2) implies (1), we simply take $s = \mathit{BOOL}$. The converse implication from (1) to (2) can be verified as follows:

$$\begin{aligned} & \mathit{cond}'\ s\ t\ t' \\ \leqslant\ & \{\text{ monotonicity of } \mathit{cond}', \text{ assumption } s \leqslant \mathit{BOOL} \} \\ & \mathit{cond}'\ \mathit{BOOL}\ t\ t' \\ \leqslant\ & \{\text{ assumption (1) }\} \\ & t \end{aligned}$$

As we shall see, formulating the constraint in terms of an upper bound on the first argument rather than an equality will ensure that the derived definition for *cond′* is indeed monotonic.

In the next step, we transform (2) into a form that removes the need for $s \leqslant \mathit{BOOL}$ as a side condition, by integrating it into the constraint itself. To achieve this, we extend the language of types with a greatest element, which we write as *TOP*:

$$\begin{tikzcd}
 & \mathit{TOP} \arrow[dl, no head] \arrow[dr, no head] & \\
\mathit{INT} \arrow[dr, no head] & & \mathit{BOOL} \arrow[dl, no head] \\
 & \mathit{ERROR} &
\end{tikzcd}$$

The new element *TOP* is only needed for the purpose of calculation. Indeed, from our specification $\mathit{texp}\ e \leqslant \mathit{tval}\ (\mathit{eval}\ e)$, it follows that *TOP* can never arise as the type of an expression, because it is not in the image of the function *tval* that returns the type of values, and hence we can never have an expression for which $\mathit{texp}\ e = \mathit{TOP}$ as this would violate the specification. Using this new element, we can now write constraint (2) in the following equivalent form:

$$\mathit{cond}'\ s\ t\ t' \quad \leqslant \quad \textbf{if}\ s \leqslant \mathit{BOOL}\ \textbf{then}\ t\ \textbf{else}\ \mathit{TOP} \tag{3}$$

In this manner, the side condition $s \leqslant \mathit{BOOL}$ is now integrated into the constraint itself. It is easy to see that (3) is equivalent to (2), because whenever $s \leqslant \mathit{BOOL}$ is false, it simplifies to $\mathit{cond}'\ s\ t\ t' \leqslant \mathit{TOP}$, which is true by definition. This form of constraint will be used often, so we find it convenient to introduce some notation to make such constraints more concise:

$$\begin{aligned}
&(\Rrightarrow) :: \mathit{Bool} \to \mathit{Type} \to \mathit{Type} \\
&b \Rrightarrow t = \textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ \mathit{TOP}
\end{aligned}$$

Using this notation, we can then write (3) simply as:

$$\mathit{cond}'\ s\ t\ t' \quad \leqslant \quad (s \leqslant \mathit{BOOL}) \Rrightarrow t \tag{4}$$

This completes the transformation process. Constraints produced by following the above steps may at first sight seem rather unusual, but have a natural operational reading. For example, the derived constraint (4) can be read as "if the first argument of a conditional could have type *BOOL*, then the conditional as a whole could have the type of the first branch." Note that the use of *could have* here rather than *has* reflects the use of inclusion $\leqslant$ rather than equality $=$.

We can apply the same transformation steps to the other constraints on *cond′* that were calculated in Section 7 to give the following equivalent set of constraints:

$$\begin{aligned}
\mathit{cond}'\ s\ t\ t' &\quad \leqslant \quad (s \leqslant \mathit{BOOL}) \Rrightarrow t \\
\mathit{cond}'\ s\ t\ t' &\quad \leqslant \quad (s \leqslant \mathit{BOOL}) \Rrightarrow t' \\
\mathit{cond}'\ s\ t\ t' &\quad \leqslant \quad (s \leqslant \mathit{INT}) \Rrightarrow \mathit{ERROR} \\
\mathit{cond}'\ s\ t\ t' &\quad \leqslant \quad (s \leqslant \mathit{ERROR}) \Rrightarrow \mathit{ERROR}
\end{aligned}$$

Collectively, these constraints state that $\mathit{cond}'\ s\ t\ t'$ is a lower bound for each of the terms on the right-hand sides. Thus, we can obtain the optimal implementation of *cond′* under these constraints by simply defining it to be the greatest such lower bound:

$$\begin{aligned}
&\mathit{cond}' :: \mathit{Type} \to \mathit{Type} \to \mathit{Type} \to \mathit{Type} \\
&\mathit{cond}'\ s\ t\ t' = ((s \leqslant \mathit{BOOL}) \Rrightarrow t) \\
&\qquad\qquad \sqcap\ ((s \leqslant \mathit{BOOL}) \Rrightarrow t') \\
&\qquad\qquad \sqcap\ ((s \leqslant \mathit{INT}) \Rrightarrow \mathit{ERROR}) \\
&\qquad\qquad \sqcap\ ((s \leqslant \mathit{ERROR}) \Rrightarrow \mathit{ERROR})
\end{aligned}$$

Taken together, this definition expresses that if the first argument of a conditional could have type *BOOL*, then the type of the result could be that of either branch, while if the type of the first argument could be *INT* or a type error, then the result could be a type error. Note that the component constraints are not disjoint, i.e. it is possible that more than one may apply, in which case the results are combined by taking their greatest lower bound.

By following a similar transformation process, we can calculate the following set of constraints for *add′* that are equivalent to those developed in Section 7:

$$\begin{aligned} add'\ t\ t' &\leqslant (t \leqslant INT \wedge t' \leqslant INT) \Rrightarrow INT \\ add'\ t\ t' &\leqslant (t \leqslant BOOL \vee t' \leqslant BOOL) \Rrightarrow ERROR \end{aligned}$$

In turn, we can immediately obtain the optimal definition of *add′* under these constraints:

$$\begin{aligned} &add' :: Type \rightarrow Type \rightarrow Type \\ &add'\ t\ t' = ((t \leqslant INT \wedge t' \leqslant INT) \quad\ \ \Rrightarrow INT) \\ &\qquad\quad \sqcap\ ((t \leqslant BOOL \vee t' \leqslant BOOL) \Rrightarrow ERROR) \end{aligned}$$

That is, if both arguments of an addition could have type *INT* then the result could have type *INT*, while if either argument could have type *BOOL* then the result could be a type error.

By using $\Rrightarrow$ and $\sqcap$ as building blocks for the definition of *cond′* and *add′*, we ensure these definitions are monotonic, as we shall consider shortly. In addition, $\Rrightarrow$ and $\sqcap$ satisfy equational laws, which we can use to simplify the definitions we have calculated:

$$\begin{aligned} (b \Rrightarrow t) \sqcap (b \Rrightarrow t') &= b \Rrightarrow (t \sqcap t') && (\Rrightarrow\text{-collect}) \\ (b \Rrightarrow t) \sqcap (b' \Rrightarrow t') &= b \Rrightarrow t \quad \text{if } b' \Rightarrow b \text{ and } t \leqslant t' && (\Rrightarrow\text{-subsume}) \\ (b \Rrightarrow t) \sqcap (b' \Rrightarrow t) &= (b \vee b') \Rrightarrow t && (\Rrightarrow\text{-disjunct}) \end{aligned}$$

For example, using $\Rrightarrow$-collect, we can simplify the first half of the definition of *cond′*:

$$((s \leqslant BOOL) \Rrightarrow t) \sqcap ((s \leqslant BOOL) \Rrightarrow t') \;=\; (s \leqslant BOOL) \Rrightarrow t \sqcap t'$$

Similarly, using $\Rrightarrow$-subsume we can simplify the second half of *cond′*:

$$((s \leqslant INT) \Rrightarrow ERROR) \sqcap ((s \leqslant ERROR) \Rrightarrow ERROR) \;=\; (s \leqslant INT) \Rrightarrow ERROR$$

Combining these two simplifications, we obtain the following equivalent definition of *cond′*:

$$\begin{aligned} &cond'\ s\ t\ t' = ((s \leqslant BOOL) \Rrightarrow t \sqcap t') \\ &\qquad\qquad \sqcap\ ((s \leqslant INT) \quad\ \Rrightarrow ERROR) \end{aligned}$$

By straightforward case analyses we can show that the definitions of *cond′* and *add′* calculated in this section are equivalent to those in Section 7. The benefit of obtaining the definitions by directly composing constraints in this manner is that the process is *entirely systematic*, and the resulting definitions are *by construction* optimal. Moreover, as we shall see in the next section, this approach will make it easier to deal with a richer language of types.

We have claimed above that the compositional definitions of *cond′* and *add′* are monotonic by construction. To be more precise, any operation $f\ x_1\ \cdots\ x_n = t$ on types is monotonic if the result type t is a constant type such *BOOL* or *INT*, one of the argument types x_i, or an arbitrary combination of these formed using $\sqcap$ and $b \Rrightarrow$, where the condition b is in turn a combination of comparisons $x_i \leqslant c$ of arguments x_i with constant types c using conjunction and disjunction.

For example, *cond′* uses the arguments t and t' along with the constant type *ERROR* and combines these using only $\sqcap$ and $\Rrightarrow$. Moreover, the conditions used by $\Rrightarrow$ only compare the argument s with constant types. A similar observation can be made for *add′*.

9 Exceptions

As a more sophisticated example of our approach to calculating type checkers, we now extend the language of conditional expressions with support for throwing and catching an exception:

```
data Expr = Val Value | Add Expr Expr | If Expr Expr Expr | Catch Expr Expr

data Value = I Int | B Bool | Throw | Error
```

Informally, the new value *Throw* represents an exception that has been thrown, while an expression *Catch x y* behaves as the expression *x* unless evaluation of *x* results in an exception being thrown, in which case the catch behaves as the *handler* expression *y*.

To define the semantics for this extended language as an evaluation function, we first extend the fold operator with a new parameter to deal with the catch primitive:

```
folde val add cond catch = f
  where
    f (Val v)       = val v
    f (Add x y)     = add (f x) (f y)
    f (If x y z)    = cond (f x) (f y) (f z)
    f (Catch x y)   = catch (f x) (f y)
```

Using this operator, we can then define an evaluation semantics for expressions by supplying the appropriate operation for each form of expression:

```
eval :: Expr → Value
eval = folde id add cond catch
```

For values, we simply use the identity function *id* as previously. Addition still requires two integers to succeed, but now also propagates an exception thrown in either argument:

```
add :: Value → Value → Value
add (I n)  (I m)  = I (n + m)
add Throw  _      = Throw
add (I _)  Throw  = Throw
add _      _      = Error
```

Note that the third clause in this definition isn't simply *add _ Throw = Throw*, as this would mean that *add Error Throw = Throw*, whereas under the normal left-to-right evaluation order for addition we expect that an error in the first argument is propagated. Conditionals are treated in the same way as previously, except that an exception thrown in the first argument is now propagated:

```
cond :: Value → Value → Value → Value
cond (B b)  v w = if b then v else w
cond Throw  _ _ = Throw
cond _      _ _ = Error
```

Finally, the new operation *catch* handles an exception thrown in the first argument by returning the second, and otherwise simply returns the value of the first argument:

```
catch :: Value → Value → Value
catch Throw v = v
catch v     _ = v
```

9.1 Type Checking

We now seek to define a type checker for our language of exceptional expressions. We begin by extending our previous language of types to include a type for an exception that has been thrown, and similarly extend the definition for the function *tval* that returns the type of a value:

$$\textbf{data}\ \mathit{Type} = \mathit{INT} \mid \mathit{BOOL} \mid \mathit{THROW} \mid \mathit{ERROR} \mid \mathit{TOP}$$

$$\begin{aligned}
&\mathit{tval} :: \mathit{Value} \rightarrow \mathit{Type}\\
&\mathit{tval}\ (I\ \_) &&= \mathit{INT}\\
&\mathit{tval}\ (B\ \_) &&= \mathit{BOOL}\\
&\mathit{tval}\ \mathit{Throw} &&= \mathit{THROW}\\
&\mathit{tval}\ \mathit{Error} &&= \mathit{ERROR}
\end{aligned}$$

The partial ordering on types is extended as shown below:

$$\begin{tikzcd}
& \mathit{TOP} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\mathit{INT} \arrow[dr, dash] & \mathit{THROW} \arrow[d, dash] & \mathit{BOOL} \arrow[dl, dash] \\
& \mathit{ERROR} &
\end{tikzcd}$$

As in Section 8, we include *TOP* as the greatest element in the language of types purely as a technical device to calculate the definition of the type checker in a compositional manner. Our goal now is to define a function $\mathit{texp} :: \mathit{Expr} \rightarrow \mathit{Type}$ that returns the type of an expression, with the desired behaviour being captured in the same manner as previously:

$$\mathit{texp}\ e \ \leqslant\ \mathit{tval}\ (\mathit{eval}\ e)$$

To calculate *tval*, we exploit the algebraic approach developed in Section 6, and assume it is defined by folding suitable operations on types, which themselves remain to be defined:

$$\mathit{texp} = \mathit{folde}\ \mathit{val}'\ \mathit{add}'\ \mathit{cond}'\ \mathit{catch}'$$

The extended fold operator also has a fusion property as previously, which requires that the new operator catch' is monotonic and satisfies a homomorphism property. Hence, by fusion, to establish $\mathit{texp}\ e \leqslant \mathit{tval}\ (\mathit{eval}\ e)$ it suffices to show that the operations add', cond' and catch' are monotonic, and the function *tval* satisfies the following homomorphism properties:

$$\begin{aligned}
\mathit{tval}\ (\mathit{id}\ x) &\ \geqslant\ \mathit{val}'\ x\\
\mathit{tval}\ (\mathit{add}\ x\ y) &\ \geqslant\ \mathit{add}'\ (\mathit{tval}\ x)\ (\mathit{tval}\ y)\\
\mathit{tval}\ (\mathit{cond}\ x\ y\ z) &\ \geqslant\ \mathit{cond}'\ (\mathit{tval}\ x)\ (\mathit{tval}\ y)\ (\mathit{tval}\ z)\\
\mathit{tval}\ (\mathit{catch}\ x\ y) &\ \geqslant\ \mathit{catch}'\ (\mathit{tval}\ x)\ (\mathit{tval}\ y)
\end{aligned}$$

We now aim to calculate definitions for the operations on types val', add', cond' and catch' that satisfy the above properties. For values, the property $\mathit{val}'\ x \leqslant \mathit{tval}\ (\mathit{id}\ x)$ simplifies immediately to $\mathit{val}'\ x \leqslant \mathit{tval}\ x$, and hence the optimal definition for val' is to simply define:

$$\begin{aligned}
&\mathit{val}' :: \mathit{Value} \rightarrow \mathit{Type}\\
&\mathit{val}'\ x = \mathit{tval}\ x
\end{aligned}$$

For addition, we proceed by case analysis on the argument values x and y. When both are integers, the calculation is the same as previously, resulting in the constraint

$$\mathit{add}'\ t\ t' \ \leqslant\ (t \leqslant \mathit{INT} \wedge t' \leqslant \mathit{INT})\ \Rrightarrow\ \mathit{INT}$$

The remaining cases lead to further constraints, as shown below:

Case: $x = \mathit{Throw}$

$$\begin{array}{ll}
& \mathit{add}'\ (\mathit{tval}\ \mathit{Throw})\ (\mathit{tval}\ y) \leqslant \mathit{tval}\ (\mathit{add}\ \mathit{Throw}\ y) \\
\Leftrightarrow & \{\text{applying } \mathit{add}, \mathit{tval}\} \\
& \mathit{add}'\ \mathit{THROW}\ (\mathit{tval}\ y) \leqslant \mathit{THROW} \\
\Leftarrow & \{\text{generalising to arbitrary type}\} \\
& \mathit{add}'\ \mathit{THROW}\ t' \leqslant \mathit{THROW} \\
\Leftrightarrow & \{\text{monotonicity of } \mathit{add}'\} \\
& \mathit{add}'\ t\ t' \leqslant \mathit{THROW}, \ \text{ if } t \leqslant \mathit{THROW} \\
\Leftrightarrow & \{\text{definition of } \Rrightarrow\} \\
& \mathit{add}'\ t\ t' \ \leqslant\ (t \leqslant \mathit{THROW}) \Rrightarrow \mathit{THROW}
\end{array}$$

Case: $x = I\ n$ and $y = \mathit{Throw}$

$$\begin{array}{ll}
& \mathit{add}'\ (\mathit{tval}\ (I\ n))\ (\mathit{tval}\ \mathit{Throw}) \leqslant \mathit{tval}\ (\mathit{add}\ (I\ n)\ \mathit{Throw}) \\
\Leftrightarrow & \{\text{applying } \mathit{add}, \mathit{tval}\} \\
& \mathit{add}'\ \mathit{INT}\ \mathit{THROW} \leqslant \mathit{tval}\ \mathit{Throw} \\
\Leftrightarrow & \{\text{applying } \mathit{tval}\} \\
& \mathit{add}'\ \mathit{INT}\ \mathit{THROW} \leqslant \mathit{THROW} \\
\Leftrightarrow & \{\text{monotonicity of } \mathit{add}'\} \\
& \mathit{add}'\ t\ t' \leqslant \mathit{THROW}, \ \text{ if } t \leqslant \mathit{INT} \text{ and } t' \leqslant \mathit{THROW} \\
\Leftrightarrow & \{\text{definition of } \Rrightarrow\} \\
& \mathit{add}'\ t\ t' \ \leqslant\ (t \leqslant \mathit{INT} \wedge t' \leqslant \mathit{THROW}) \Rrightarrow \mathit{THROW}
\end{array}$$

Case: $x = \mathit{Error}$ or $x = B\ b$ or $(x = I\ n$ and $(y = \mathit{Error}$ or $y = B\ b))$

$$\begin{array}{ll}
& \mathit{add}'\ (\mathit{tval}\ x)\ (\mathit{tval}\ y) \leqslant \mathit{tval}\ (\mathit{add}\ x\ y) \\
\Leftrightarrow & \{\text{applying } \mathit{add}, \mathit{tval}\} \\
& \mathit{add}'\ (\mathit{tval}\ x)\ (\mathit{tval}\ y) \leqslant \mathit{ERROR} \\
\Leftarrow & \{\text{generalising to arbitrary types}\} \\
& \mathit{add}'\ t\ t' \leqslant \mathit{ERROR}, \ \text{ if } t = \mathit{ERROR} \text{ or } t = \mathit{BOOL} \text{ or } (t = \mathit{INT} \text{ and } (t' = \mathit{ERROR} \text{ or } t' = \mathit{BOOL})) \\
\Leftrightarrow & \{\text{monotonicity of } \mathit{add}'\} \\
& \mathit{add}'\ t\ t' \leqslant \mathit{ERROR}, \ \text{ if } t \leqslant \mathit{ERROR} \text{ or } t \leqslant \mathit{BOOL} \text{ or } (t \leqslant \mathit{INT} \text{ and } (t' \leqslant \mathit{ERROR} \text{ or } t' \leqslant \mathit{BOOL})) \\
\Leftrightarrow & \{\text{simplifying, } \mathit{ERROR} \leqslant \mathit{BOOL}\} \\
& \mathit{add}'\ t\ t' \leqslant \mathit{ERROR}, \ \text{ if } t \leqslant \mathit{BOOL} \text{ or } (t \leqslant \mathit{INT} \text{ and } t' \leqslant \mathit{BOOL}) \\
\Leftrightarrow & \{\text{definition of } \Rrightarrow\} \\
& \mathit{add}'\ t\ t' \ \leqslant\ (t \leqslant \mathit{BOOL} \vee (t \leqslant \mathit{INT} \wedge t' \leqslant \mathit{BOOL})) \Rrightarrow \mathit{ERROR}
\end{array}$$

The final case above covers the remaining argument possibilities not covered by the other cases. It is formulated in a positive manner by explicitly enumerating the remaining possibilities. This ensures that the calculated definition of *add′* is monotonic by construction.

In conclusion, the above reasoning shows that the following four constraints for add' are together sufficient to satisfy the homomorphism property for addition:

$$\begin{array}{lcl}
add'\ t\ t' & \leqslant & (t \leqslant INT \wedge t' \leqslant INT) \Rrightarrow INT \\
add'\ t\ t' & \leqslant & (t \leqslant THROW) \Rrightarrow THROW \\
add'\ t\ t' & \leqslant & (t \leqslant INT \wedge t' \leqslant THROW) \Rrightarrow THROW \\
add'\ t\ t' & \leqslant & (t \leqslant BOOL \vee (t \leqslant INT \wedge t' \leqslant BOOL)) \Rrightarrow ERROR
\end{array}$$

The optimal solution to these constraints is then obtained by defining the add' operation as the greatest lower bound of each of the right-hand sides:

$$\begin{array}{lll}
add'\ t\ t' = & ((t \leqslant INT \wedge t' \leqslant INT) & \Rrightarrow INT) \\
\sqcap & ((t \leqslant THROW) & \Rrightarrow THROW) \\
\sqcap & ((t \leqslant INT \wedge t' \leqslant THROW) & \Rrightarrow THROW) \\
\sqcap & ((t \leqslant BOOL \vee (t \leqslant INT \wedge t' \leqslant BOOL)) & \Rrightarrow ERROR)
\end{array}$$

Using the $\Rrightarrow$-disjunct law, we can then combine the second and third component to give a single constraint that returns the type $THROW$, resulting in the following final definition:

$$\begin{array}{lll}
\multicolumn{3}{l}{add' :: Type \rightarrow Type \rightarrow Type} \\
add'\ t\ t' = & ((t \leqslant INT \wedge t' \leqslant INT) & \Rrightarrow INT) \\
\sqcap & ((t \leqslant THROW \vee (t \leqslant INT \wedge t' \leqslant THROW)) & \Rrightarrow THROW) \\
\sqcap & ((t \leqslant BOOL \vee (t \leqslant INT \wedge t' \leqslant BOOL)) & \Rrightarrow ERROR)
\end{array}$$

For conditionals, there is only one new case to consider, which proceeds as follows:

Case: $x = Throw$

$$\begin{array}{ll}
 & cond'\ (tval\ Throw)\ (tval\ y)\ (tval\ z) \leqslant tval\ (cond\ Throw\ y\ z) \\
\Leftrightarrow & \{\text{applying } cond,\ tval\} \\
 & cond'\ THROW\ (tval\ y)\ (tval\ z) \leqslant THROW \\
\Leftarrow & \{\text{generalising to arbitrary types}\} \\
 & cond'\ THROW\ t\ t' \leqslant THROW \\
\Leftrightarrow & \{\text{monotonicity of } cond'\} \\
 & cond'\ s\ t\ t' \leqslant THROW, \quad \text{if } s \leqslant THROW \\
\Leftrightarrow & \{\text{definition of } \Rrightarrow\} \\
 & cond'\ s\ t\ t' \ \leqslant\ (s \leqslant THROW) \Rrightarrow THROW
\end{array}$$

Together with the four constraints we already calculated in Section 8, we thus obtain the following five constraints for $cond'$, which together ensure the required homomorphism property:

$$\begin{array}{lcl}
cond'\ s\ t\ t' & \leqslant & (s \leqslant BOOL) \Rrightarrow t \\
cond'\ s\ t\ t' & \leqslant & (s \leqslant BOOL) \Rrightarrow t' \\
cond'\ s\ t\ t' & \leqslant & (s \leqslant INT) \Rrightarrow ERROR \\
cond'\ s\ t\ t' & \leqslant & (s \leqslant ERROR) \Rrightarrow ERROR \\
cond'\ s\ t\ t' & \leqslant & (s \leqslant THROW) \Rrightarrow THROW
\end{array}$$

We can then immediately obtain the optimal definition of $cond'$ that satisfies these constraints. As in Section 8, we can then simplify the resulting definition by combining the first and second component as well as the third and fourth, to give the following definition:

$$\begin{aligned}
&cond' :: Type \to Type \to Type \to Type\\
&cond'\ s\ t\ t' = ((s \leqslant BOOL) \Rrightarrow t \sqcap t')\\
&\qquad\quad \sqcap\ ((s \leqslant INT) \Rrightarrow ERROR)\\
&\qquad\quad \sqcap\ ((s \leqslant THROW) \Rrightarrow THROW)
\end{aligned}$$

Finally, for the catch operation that handles exceptions there are two cases to consider, namely whether the first argument results in an exception being thrown or not:

Case: $x = Throw$

$$\begin{aligned}
&\quad catch'\ (tval\ Throw)\ (tval\ y) \leqslant tval\ (catch\ Throw\ y)\\
\Leftrightarrow&\quad \{\text{ applying } catch, tval\ \}\\
&\quad catch'\ THROW\ (tval\ y) \leqslant tval\ y\\
\Leftarrow&\quad \{\text{ generalising to arbitrary type }\}\\
&\quad catch'\ THROW\ t' \leqslant t'\\
\Leftrightarrow&\quad \{\text{ monotonicity of } catch'\ \}\\
&\quad catch'\ t\ t' \leqslant t', \text{ if } t \leqslant THROW\\
\Leftrightarrow&\quad \{\text{ definition of } \Rrightarrow\ \}\\
&\quad catch'\ t\ t' \leqslant (t \leqslant THROW) \Rrightarrow t'
\end{aligned}$$

Case: $x = Error$ or $x = B\ b$ or $x = I\ n$

$$\begin{aligned}
&\quad catch'\ (tval\ x)\ (tval\ y) \leqslant tval\ (catch\ x\ y)\\
\Leftrightarrow&\quad \{\text{ applying } catch\ \}\\
&\quad catch'\ (tval\ x)\ (tval\ y) \leqslant tval\ x\\
\Leftarrow&\quad \{\text{ generalising to arbitrary types }\}\\
&\quad catch'\ t\ t' \leqslant t, \text{ if } t = ERROR \text{ or } t = BOOL \text{ or } t = INT\\
\Leftrightarrow&\quad \{\text{ monotonicity of } catch'\ \}\\
&\quad catch'\ t\ t' \leqslant t'', \text{ if } t \leqslant t'' \text{ and } t'' \in \{ERROR, BOOL, INT\}\\
\Leftrightarrow&\quad \{\text{ definition of } \Rrightarrow\ \}\\
&\quad catch'\ t\ t' \leqslant (t \leqslant t'') \Rrightarrow t'', \text{ if } t'' \in \{ERROR, BOOL, INT\}
\end{aligned}$$

The above reasoning shows that the following four constraints for *catch'* are together sufficient to satisfy the homomorphism property for catching exceptions:

$$\begin{aligned}
catch'\ t\ t' &\leqslant (t \leqslant THROW) \Rrightarrow t'\\
catch'\ t\ t' &\leqslant (t \leqslant ERROR) \Rrightarrow ERROR\\
catch'\ t\ t' &\leqslant (t \leqslant BOOL) \Rrightarrow BOOL\\
catch'\ t\ t' &\leqslant (t \leqslant INT) \Rrightarrow INT
\end{aligned}$$

From this we immediately obtain the optimal definition for *catch'*:

$$\begin{aligned}
&catch' :: Type \to Type \to Type\\
&catch'\ t\ t' = ((t \leqslant THROW) \Rrightarrow t')\\
&\qquad\quad \sqcap\ ((t \leqslant ERROR) \Rrightarrow ERROR)\\
&\qquad\quad \sqcap\ ((t \leqslant BOOL) \Rrightarrow BOOL)\\
&\qquad\quad \sqcap\ ((t \leqslant INT) \Rrightarrow INT)
\end{aligned}$$

In summary, using the techniques developed in previous sections, we have calculated a type checker for the language of exceptional expressions. Moreover, each of the derived operations on types

add', $cond'$ and $catch'$ is built using the pattern described in Section 8 and is hence monotonic by construction, so there is no need to manually check this property.

9.2 More Informative Types

The type checker developed in the previous section is correct, as it satisfies the specification $texp\ e \leqslant tval\ (eval\ e)$, but is less informative than we might wish. In this section we explain the problem, and show how it can be solved by further exploiting algebraic properties.

The problem is with how exceptions are treated in conditional expressions. In particular, if we have a conditional where one branch produces a regular value and the other branch throws an exception, this will be regarded as being ill-typed. For example, if we represent the expression **if** *True* **then** 1 **else** *Throw* in our language, then applying the type checking function *texp* will give the result *ERROR*, meaning that the expression contains a type error. This behaviour arises from the definition of the conditional operation on types:

$$
\begin{array}{lll}
cond'\ s\ t\ t' = & ((s \leqslant \mathit{BOOL}) & \Rrightarrow t \sqcap t') \\
\sqcap & ((s \leqslant \mathit{INT}) & \Rrightarrow \mathit{ERROR}) \\
\sqcap & ((s \leqslant \mathit{THROW}) & \Rrightarrow \mathit{THROW})
\end{array}
$$

Using this definition, the example expression has type

$$cond'\ \mathit{BOOL}\ \mathit{INT}\ \mathit{THROW} \;=\; \mathit{INT} \sqcap \mathit{THROW} \;=\; \mathit{ERROR}$$

In contrast, in Haskell the similar expression **if** *True* **then** 1 **else** *throw SomeException* type checks fine as an integer. In Haskell, this is achieved by giving the function *throw* a polymorphic type, namely $throw :: Exception\ e \Rightarrow e \rightarrow a$. However, this approach might be viewed as unsatisfactory, as a polymorphic result type gives no indication that an exception may be thrown. It would be preferable to solve the problem by making the type system more informative.

To deal with exceptions in a more informative way, the type system needs to be revised so that $\mathit{INT} \sqcap \mathit{THROW}$ is not simply *ERROR*. This can be achieved by adding a new type, say *INT*?, that represents a value that could be either an integer or an exception, and similarly, a new type *BOOL*? that represents a value that could be either a logical value or an exception:

data *Type* = *INT* | *INT*? | *BOOL* | *BOOL*? | *THROW* | *ERROR* | *TOP*

The desired behaviour of the new types is specified by the equations $\mathit{INT} \sqcap \mathit{THROW} = \mathit{INT}?$ and $\mathit{BOOL} \sqcap \mathit{THROW} = \mathit{BOOL}?$, which can be realised by refining the ordering on types to:

$$\begin{tikzcd}
 & & \mathit{TOP} \arrow[dll, dash] \arrow[d, dash] \arrow[drr, dash] & & \\
\mathit{INT} \arrow[dr, dash] & & \mathit{THROW} \arrow[dl, dash] \arrow[dr, dash] & & \mathit{BOOL} \arrow[dl, dash] \\
 & \mathit{INT}? \arrow[dr, dash] & & \mathit{BOOL}? \arrow[dl, dash] & \\
 & & \mathit{ERROR} & &
\end{tikzcd}$$

In this manner, we now have an extra level of types between the error type and the original value types, which allow us to represent values that could potentially be exceptions.

We now consider what further changes are required. First of all, because the type of values is unchanged, the function $tval :: Value \rightarrow Type$ does not require any modification. In turn, calculations of the constraints for the type operations val', add', $cond'$ and $catch'$ also remain unchanged, because the calculations only use positive facts about the ordering, and any ordering on types $t \leqslant t'$ in the original definition of *Type* still holds in the extended definition. Finally, because we defined *val′*, *add′*, *cond′* and *catch′* as optimal solutions for the calculated constraints, these definitions also satisfy the same constraints for the extended type, and are still the optimal solution. Hence, the definition of the type checker $texp :: Expr \rightarrow Type$ can also remain unchanged.

In summary, due to the manner in which the type checker was constructed in the previous section, no changes are required beyond modifying the language of types. That is, the existing definitions can be used without modification with the extended notion of types. Crucially, however, the type checker can now give more informative results. For example, type checking the expression **if** *True* **then** 1 **else** *Throw* now gives the following result:

$$cond'\ BOOL\ INT\ THROW \quad = \quad INT \sqcap THROW \quad = \quad INT?$$

That is, the type checker now indicates that the expression could either produce an integer or raise an exception, whereas under the previous definition it simply resulted in a type error.

As another example, consider the expression *catch* (**if** *True* **then** 1 **else** *Throw*) 2. As shown above, the conditional expression has type *INT*?, hence the whole expression has type:

$$catch'\ INT?\ INT \quad = \quad INT \sqcap INT \quad = \quad INT$$

In this manner, the type checker recognises that catch allows us to recover from the *possibility* of an exception being raised by the conditional and thereby guarantee to return an integer result.

10 Lambda Calculus

As a final example, we extend the expression language from Section 3 with lambda abstraction and application. This requires a further generalisation to our methodology, to deal with the extra complexities of the language. In doing so, we demonstrate that our techniques extend naturally to more realistic languages with support for variables and bindings.

We use higher-order abstract syntax [Pfenning and Elliott 1988] to represent lambda abstractions, as doing so keeps the semantics simple and avoids the need for variable environments:

```
data Expr = Val Value | Add Expr Expr | If Expr Expr Expr
          | Abs (Expr → Expr) | App Expr Expr
```

For example, the expression *App* (*Abs* ($\lambda x \rightarrow Add\ x\ x$)) (*Val* (*I* 1)) applies a doubling function to an integer. As usual with higher-order abstract syntax, in an abstraction *Abs f* the function *f* must be *parametric*, treating its argument purely symbolically and not performing case analysis on it.

For simplicity, we consider a first-order version of the lambda calculus in which abstractions cannot be passed as arguments in applications. This choice ensures that evaluation always terminates, and hence we can use the following type as our semantic domain:

```
data Value = I Int | B Bool | Fn (Value → Value) | Error
```

Again, this extends naturally from Section 3. The new value constructor *Fn* represents closures, with the function argument capturing the semantic behaviour of an abstraction on values.

Enforcing that the language is first-order is achieved within the semantics, which as previously is expressed by a function of type $Expr \rightarrow Value$ defined using a fold operator. However, defining a suitable fold operator for the lambda calculus language is more involved, as *Expr* appears both positively and negatively within the *Abs* constructor. In practice, this mixed variance use of the expression type means that structural recursion alone is insufficient.

Fortunately, this issue is explored by Meijer and Hutton [1995] and Fegaras and Sheard [1996]. The former presents a solution, defining folds mutually with *unfolds* to handle the mixed variance.

```
folde val add cond abs app undo = f
  where
```

$$\begin{aligned}
&f\ (Val\ v) &&= val\ v\\
&f\ (Add\ x\ y) &&= add\ (f\ x)\ (f\ y)\\
&f\ (If\ x\ y\ z) &&= cond\ (f\ x)\ (f\ y)\ (f\ z)\\
&f\ (Abs\ k) &&= abs\ (f \circ k \circ Val \circ undo)\\
&f\ (App\ x\ y) &&= app\ (f\ x)\ (f\ y)
\end{aligned}$$

In our setting, we only require unfolds which produce values, so we have simplified the setup by defining the fold in terms of a function $undo :: a \to Value$. Then we use the unfold, defined by $Val \circ undo$, as part of the fold over abstractions. The semantics can then be defined by:

$$\begin{aligned}
&eval :: Expr \to Value\\
&eval = folde\ id\ add\ cond\ abs\ app\ id
\end{aligned}$$

The operations for values, addition and conditionals are the same as in Section 3, while the new operations for abstractions and applications are defined by:

$$\begin{aligned}
&abs :: (Value \to Value) \to Value\\
&abs = Fn
\end{aligned}$$

$$\begin{aligned}
&app :: Value \to Value \to Value\\
&app\ (Fn\ f)\ (I\ n) &&= f\ (I\ n)\\
&app\ (Fn\ f)\ (B\ b) &&= f\ (B\ b)\\
&app\ \_\qquad\ \ \_ &&= Error
\end{aligned}$$

Note that *app* ensures the language is first-order by only allowing integers and logical values as function arguments. Finally, *undo* is simply the identity function, as we already have a value. This gives $eval\ (Abs\ f) = Fn\ (eval \circ f \circ Val)$, in which the body of *Fn* lifts a value to an expression, *syntactically* transforms it via the parametric function f, and then evaluates it back to a value.

Now we move on to type-checking. As previously, we start with a language of types that includes one constructor for each form of value, plus a *TOP* element:

data $Type = INT \mid BOOL \mid FN \mid ERROR \mid TOP$

However, defining type checking as a fold is now problematic. In particular, we require an operation $undo' :: Type \to Value$, but it is not clear how it should be defined. For example, what should we return for $undo'\ INT$? Following Fegaras and Sheard [1996], we solve this problem by adding a new value constructor, called T, which allows us to abstractly *reify* types into values:

data $Value = I\ Int \mid B\ Bool \mid Fn\ (Value \to Value) \mid Error \mid T\ Type$

We define $tval\ (T\ \ t) = t$. Now, the constructor $T :: Type \to Value$ can serve as $undo'$. Unfortunately, our semantics is not equipped to suitably deal with T values. For example, at present $add\ (T\ INT)\ (I\ 1)$ simply gives an error value, due to the wildcard case $add\ \_\ \_ = Error$.

Recall, however, that types are ordered, which induces a partial ordering $\leqslant$ on values:

instance *Ord Value* **where**

$$\begin{aligned}
&\quad (\leqslant) :: Value \to Value \to Bool\\
&\quad v \leqslant T\ t = tval\ v \leqslant t\\
&\quad v \leqslant w\ \ \ = v == w
\end{aligned}$$

This ordering provides us with two properties that we will exploit in our calculation: First, it establishes a Galois connection between types and values. Secondly, we require that the operations on values *add*, *cond* and so on preserve the ordering on values, i.e. they are monotonic. This monotonicity requirement also allows us to derive the missing cases for T values in the definitions of *add*, *cond* etc. For example, consider the clause $add\ (I\ n)\ (I\ m) = I\ (n+m)$ for addition. Monotonicity requires that if $I\ n \leqslant v$ and $I\ m \leqslant w$ then $add\ (I\ n)\ (I\ m) \leqslant add\ v\ w$, from which we can easily deduce the following additional cases for *add*:

$$\begin{array}{l}
add\ (T\ INT)\ (I\ \_) \quad\ \ = T\ INT \\
add\ (I\ \_) \qquad\ (T\ INT) = T\ INT \\
add\ (T\ INT)\ (T\ INT) = T\ INT
\end{array}$$

That is, reified integer types are propagated, which is natural as they represent any integer value. Similar reasoning for *cond*, *abs* and *app* provides a complete semantics for values that arise by reifying types. For example, we can deduce that $cond\ (T\ BOOL)\ v\ w = T\ (tval\ v \sqcap tval\ w)$.

The new fold operator satisfies a fusion property as previously, which in the case of abstraction has a condition that must hold for any function $f :: Expr \to a$:

$$\begin{array}{rcl}
h\ (val\ x) & \geqslant & val'\ x \\
h\ (add\ x\ y) & \geqslant & add'\ (h\ x)\ (h\ y) \\
h\ (cond\ x\ y\ z) & \geqslant & cond'\ (h\ x)\ (h\ y)\ (h\ z) \\
h\ (abs\ (f \circ Val \circ undo)) & \geqslant & abs'\ (h \circ f \circ Val \circ undo') \\
h\ (app\ x\ y) & \geqslant & app'\ (h\ x)\ (h\ y) \\
\hline
h\ (folde\ val\ add\ cond\ abs\ app\ undo\ e) & \geqslant & folde\ val'\ add'\ cond'\ abs'\ app'\ undo'\ e
\end{array}$$

Fusion also requires that add', $cond'$, abs', app' are monotonic. We now seek to calculate *texp* from the specification $tval\ (eval\ e) \geqslant texp\ e$, using the assumption that *texp* is defined by folding suitable operations on types. The cases for values, addition and conditionals are similar to those in Section 3, so we begin with application. The first case, when $x = Fn\ f$ and $y = I\ n$, begins as follows:

$$\begin{array}{ll}
& app'\ (tval\ (Fn\ f))\ (tval\ (I\ n)) \leqslant tval\ (app\ (Fn\ f)\ (I\ n)) \\
\Leftrightarrow & \{\text{ applying } app,\ tval\ \} \\
& app'\ (tval\ (Fn\ f))\ INT \leqslant tval\ (f\ (I\ n)) \\
\Leftrightarrow & \{\text{ monotonicity of } tval \text{ and } f\ \} \\
& app'\ (tval\ (Fn\ f))\ INT \leqslant tval\ (f\ (T\ INT))
\end{array}$$

Monotonicity of *tval* follows from the value-type Galois connection. Moreover, for any result $Fn\ f$ of evaluating $Abs\ g$, we can show that f is monotonic from the assumption that g is parametric and that *add*, *cond*, *abs*, *app* are monotonic. We may thus assume monotonicity of f above.

But now we seem to be stuck. In particular, if we were to view the above as a definition for app', the right-hand side would require us to know $tval\ (f\ (T\ INT))$, but $tval\ (Fn\ f)$ is simply *FN*, giving us no such information. However, we can take a hint from similar situations arising in compiler calculation [Bahr and Hutton 2015], and carry the required additional information within the constructor. That is, if we extend the *FN* constructor to hold a *Type* argument, we can proceed:

$$\begin{array}{ll}
& app'\ (tval\ (Fn\ f))\ INT \leqslant tval\ (f\ (T\ INT)) \\
\Leftrightarrow & \{\text{ redefine: } tval\ (Fn\ f) = FN\ (tval\ (f\ (T\ INT)))\ \} \\
& app'\ (FN\ (tval\ (f\ (T\ INT))))\ INT \leqslant tval\ (f\ (T\ INT)) \\
\Leftarrow & \{\text{ generalise to arbitrary type }\} \\
& app'\ (FN\ t)\ INT \leqslant t \\
\Leftrightarrow & \{\text{ monotonicity of } app'\ \} \\
& app'\ (FN\ t)\ u \leqslant t, \quad \text{if } u \leqslant INT \\
\Leftrightarrow & \{\text{ definition of } \Rrightarrow\ \} \\
& app'\ (FN\ t)\ u\ \leqslant\ (u \leqslant INT)\ \Rrightarrow\ t
\end{array}$$

Playing the same trick with the second case for application, $x = Fn\ f$ and $y = B\ b$, we discover that we need a second *Type* argument for *FN*, with the type of function values then defined by:

$$tval\ (Fn\ f) = FN\ (tval\ (f\ (T\ INT)))\ (tval\ (f\ (T\ BOOL)))$$

In summary, we can derive the following two initial constraints for application:

$$\begin{aligned} app'\ (FN\ t\ t')\ u &\leqslant (u \leqslant INT) \Rrightarrow t \\ app'\ (FN\ t\ t')\ u &\leqslant (u \leqslant BOOL) \Rrightarrow t' \end{aligned}$$

Due to the lack of polymorphism, the function type $FN\ t\ t'$ is essentially an intersection type $(INT \to t, BOOL \to t')$, which separately describes a function's behaviour when applied to an integer or logical value, respectively. Hence, the two constraints above express that if the argument of a function application could have type *INT*, then the application as a whole could have the type for the first argument of *FN*, and similarly, if the argument could have type *BOOL*, then the application could have the type of the second argument of *FN*.

The third constraint for application arises by considering $x = Fn\ f$ and $y = Fn\ g$, and utilising the pointwise ordering $FN\ t\ t' \leqslant FN\ u\ u'$ iff $t \leqslant u$ and $t' \leqslant u'$, from which we obtain:

$$app'\ (FN\ t\ t')\ u \leqslant (u \leqslant FN\ TOP\ TOP) \Rrightarrow ERROR$$

Under the pointwise ordering, this constraint expresses that if the argument of a function application could have a function type, then the application could have a type error. The final case for application, when $x \neq Fn\ f$, is straightforward, and yields the following constraint:

$$app'\ t\ u \leqslant ERROR \qquad \text{if } t \in \{INT, BOOL, ERROR, TOP\}$$

Additional defining clauses of *app*, such as $app\ (Fn\ f)\ (T\ INT)$, which we obtained from the value ordering, do not result in any additional constraints. They are subsumed automatically by the above four. This convenient general fact also holds for the other cases too.

We now move on to abstraction. Because *abs* only has one defining equation, only one case is needed, and the calculation proceeds by simply unfolding definitions:

$$\begin{aligned} & abs'\ (tval \circ f \circ Val \circ T) \leqslant tval\ (abs\ (f \circ Val \circ id)) \\ \Leftrightarrow\quad & \{\text{applying } abs, id\} \\ & abs'\ (tval \circ f \circ Val \circ T) \leqslant tval\ (Fn\ (f \circ Val)) \\ \Leftrightarrow\quad & \{\text{applying of } tval\} \\ & abs'\ (tval \circ f \circ Val \circ T) \leqslant FN\ (tval\ (f\ (Val\ (T\ INT))))\ (tval\ (f\ (Val\ (T\ BOOL)))) \\ \Leftarrow\quad & \{\text{generalising to arbitrary function on types}\} \\ & abs'\ g \leqslant FN\ (g\ INT)\ (g\ BOOL) \end{aligned}$$

Finally, we can then compose the constraints that we have derived for each of the operations on types, resulting in the following general definitions:

$$\begin{aligned} & add' :: Type \to Type \to Type \\ & add'\ t\ t' = ((t \leqslant INT \land t' \leqslant INT) \Rrightarrow INT) \\ & \qquad\quad \sqcap ((t \leqslant BOOL \lor t' \leqslant BOOL \lor t \leqslant FN\ TOP\ TOP \lor t' \leqslant FN\ TOP\ TOP) \Rrightarrow ERROR) \end{aligned}$$

$$\begin{aligned} & cond' :: Type \to Type \to Type \to Type \\ & cond'\ t\ u\ u' = ((t \leqslant BOOL) \Rrightarrow u \sqcap u') \\ & \qquad\qquad \sqcap ((t \leqslant INT \lor t \leqslant FN\ TOP\ TOP) \Rrightarrow ERROR) \end{aligned}$$

$$\begin{aligned} & abs' :: (Type \to Type) \to Type \\ & abs'\ f = FN\ (f\ INT)\ (f\ BOOL) \end{aligned}$$

$$
\begin{array}{lll}
\mathit{app}' :: \mathit{Type} \to \mathit{Type} \to \mathit{Type} & & \\
\mathit{app}'\ (\mathit{FN}\ t\ t')\ u = & ((u \leqslant \mathit{INT}) & \Rrightarrow t) \\
 & \sqcap\ ((u \leqslant \mathit{BOOL}) & \Rrightarrow t') \\
 & \sqcap\ ((u \leqslant \mathit{FN\ TOP\ TOP}) & \Rrightarrow \mathit{ERROR}) \\
\mathit{app}'\ t \qquad\quad u = \mathit{ERROR} & &
\end{array}
$$

In summary, we have demonstrated how our methodology extends to more complex languages with variables and binders. To achieve this we used a more powerful fold operator and associated fusion property, together with an ordering on values as well as types. We also exploited the idea of extending the language of types during the calculation process, inspired by a similar approach that has been used in previous work on compiler calculation.

11 Related Work

The idea of calculating a type checker from a single inequality is inspired by the compiler calculation methodology of Bahr and Hutton [2015]. This approach starts with specification that expresses the correctness of a compiler as a single equation relating the semantics of source programs with the target code. From this specification the compiler is then calculated using equational reasoning techniques. Similarly to the type checker calculation presented in this article, later versions of this compiler calculation approach also used an inequation as the specification [Bahr and Hutton 2020]. However, the calculation of type checkers has required further refinement to structure the calculation: using an ordered version of fold fusion to calculate inequality constraints, and a means to obtain an optimal definition for these constraints in a compositional manner.

A type checker is a special case of an *abstract interpreter* [Cousot 1997]. The soundness of an abstract interpreter f_a with respect to a corresponding concrete interpreter f_c is typically formulated by a Galois connection, which consists of two inequalities, $f_a \circ \alpha \leqslant \alpha \circ f_c$ and $\gamma \circ f_a \leqslant f_c \circ \gamma$, for suitable abstraction and concretisation functions α and γ. In our setting, the concrete interpreter f_c is *eval*, the abstract interpreter f_a is the type checker *texp*, and the abstraction function α is *tval*. Our methodology dispenses with the concretisation function γ, and also the second inequality. The remaining single inequality suffices to capture the desired soundness property of the type checker. While this simplifies the calculation process, it remains to be seen whether our approach based on a single inequality generalises to non-terminating and higher-order languages. However, there is reason for optimism, in particular because Bahr and Hutton [2015] have shown that the compiler calculation technique of Meijer [1992] based on an adjunction, a generalisation of Galois connections, can be simplified to a single equation by transforming the higher-order semantics of the source language into first-order form by defunctionalisation.

Our calculation of a type checker for a language with exceptions naturally led to a type system with *checked exceptions* [Goodenough 1975; Liskov and Snyder 1979], a type system feature popularised in Java [Gosling et al. 1996]. Exceptions and exception handling are a special case of *algebraic effects* and *effect handling* [Plotkin and Pretnar 2009]. Type and effect systems have been proposed to check for the presence and proper handling of effects [Bauer and Pretnar 2013]. This suggests further work to explore how such type and effects systems for algebraic effects can be calculated using the kind of techniques presented in this article.

12 Conclusion and Further Work

In this article, we showed how the calculational approach to program construction can be applied to the design of type checkers. In particular, starting from a specification that captures the desired behaviour of a type checker, we showed how equational reasoning techniques can be used to derive a correct-by-construction implementation in a principled, systematic manner.

There are a number of possible avenues for further work. First of all, the language of types was provided up front as part of the specification, but it would be interesting to calculate the types at the same time as the type checker, in a similar manner to how Bahr and Hutton [2015] calculate the language of code at the same time as a compiler. Indeed, we already touched on this idea in the lambda calculus example in Section 10. Secondly, it is important to consider how the methodology scales to more sophisticated source language and type system features, such as computational effects, type constructors, and polymorphism. Thirdly, it would be interesting to explore how the calculations can be formalised in a proof assistant, but based on previous experience we do not expect this to be problematic. And finally, it would be interesting to calculate typing rules as opposed to type checkers, and we already have some promising results in this area.

Acknowledgements

This work was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/Y010744/1, *Semantics-Directed Compiler Construction: From Formal Semantics to Certified Compilers*, for which funding is gratefully acknowledged.

References

Roland Backhouse. 2003. *Program Construction: Calculating Implementations from Specifications.* John Wiley and Sons, Inc.

Patrick Bahr and Graham Hutton. 2015. Calculating Correct Compilers. *Journal of Functional Programming* 25 (2015).

Patrick Bahr and Graham Hutton. 2020. Calculating Correct Compilers II: Return of the Register Machines. *Journal of Functional Programming* 30 (2020).

Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science.* Springer Berlin Heidelberg.

Patrick Cousot. 1997. Types as Abstract Interpretations. In *Proceedings of the Symposium on Principles of Programming Languages.* ACM, Paris, France.

Leonidas Fegaras and Tim Sheard. 1996. Revisiting Catamorphisms over Datatypes with Embedded Functions (or, Programs from Outer Space). In *Proceedings of the Symposium on Principles of programming languages.*

John B. Goodenough. 1975. Exception Handling: Issues and a Proposed Notation. *Commun. ACM* 18, 12 (1975).

James Gosling, Bill Joy, and Guy L. Steele. 1996. *The Java Language Specification.* Addison-Wesley Longman Publishing.

Barbara H. Liskov and Alan Snyder. 1979. Exception Handling in CLU. *IEEE Transactions on Software Engineering* 5, 6 (1979).

Simon Marlow et al. 2010. Haskell 2010 Language Report. *Available online at https://www.haskell.org* (2010).

Erik Meijer. 1992. *Calculating Compilers.* PhD Thesis. Katholieke Universiteit Nijmegen.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the Conference on Functional Programming and Computer Architecture.*

Erik Meijer and Graham Hutton. 1995. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture.*

Flemming Nielson, Hanne Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer.

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. *ACM Sigplan Notices* 23, 7 (1988).

Benjamin C. Pierce. 2002. *Types and Programming Languages.* The MIT Press.

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems (Lecture Notes in Computer Science, Vol. 5502).* Springer Berlin Heidelberg.