The Calculated Typer (Functional Pearl)

Zac Garby
University of Nottingham
United Kingdom
zac.garby@nottingham.ac.uk

Patrick Bahr
IT University of Copenhagen
Denmark
paba@itu.dk

Graham Hutton
University of Nottingham
United Kingdom
graham.hutton@nottingham.ac.uk

Abstract

We present a calculational approach to the design of type checkers, showing how they can be derived from behavioural specifications using equational reasoning. We focus on languages whose semantics can be expressed as a fold, and show how the calculations can be simplified using fold fusion. This approach enables the compositional derivation of correct-by-construction type checkers based on solving and composing fusion preconditions. We introduce our approach using a simple expression language, to which we then add support for exception handling and checked exceptions.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Logic and verification**; **Type theory**; **Program verification**.

Keywords: program calculation, type checking

ACM Reference Format:
Zac Garby, Patrick Bahr, and Graham Hutton. 2025. The Calculated Typer (Functional Pearl). In *Proceedings of the 18th ACM SIGPLAN International Haskell Symposium (Haskell '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3759164.3759346

1 Introduction

Type checking ensures that programs are constructed in accordance with a set of typing rules [14]. 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 + *True* is ill typed because the second argument is not an integer. Moreover, we can say that 1 + 2 has integer type because it produces an integer result, while 1 + *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 whether it is well typed. If the 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 predetermined 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 that 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 in some way. 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 a number of benefits. First of all, the resulting type checkers are *correct-by-construction* [1], eliminating the need for separate correctness proofs. Secondly, the approach provides a systematic way to *discover* typing rules, and to explore alternative design choices during the calculation process. And finally, it is readily amenable to mechanical *formalisation*, enabling the use of proof assistants as interactive tools for developing and certifying the calculations.

Our goal is not to devise a general-purpose approach to designing type systems for full-featured programming languages. Instead, we explore the underlying algebraic structure of type checking, and exploit this structure to enable the derivation of correct-by-construction type checkers using simple, compositional reasoning principles. To this end, we focus on terminating, first-order languages.

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

Haskell '25, Singapore, Singapore

ACM ISBN 979-8-4007-2147-2/25/10
https://doi.org/10.1145/3759164.3759346

composing fusion preconditions. We introduce our approach using a simple expression language with conditionals, then extend the language with support for exception handling, and finally consider checked exceptions.

All our programs are written in Haskell [10], but the ideas are applicable in any functional programming language. Because the calculations are the central focus, they are typically presented in detail rather than being compressed or omitted. Haskell code for each of the examples, together with an Agda formalisation of all of the main results, is freely available online as supplementary material [6].

2 Positivity Checking

Type checking is an example of static analysis [13], 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:

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

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

Suppose now that we wish to define a function $\mathit{isPos} :: \mathit{Expr} \to \mathit{Bool}$ 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. Each step in the calculation is either a logical equivalence ($\Leftrightarrow$) or a reverse implication ($\Leftarrow$), so that the final term of the calculation, $\mathit{isPos}\ e$, implies the term that we started with, $\mathit{eval}\ e > 0$.

Case: $\mathit{Val}\ n$

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

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

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

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:

$$\begin{aligned}&\mathit{isPos} :: \mathit{Expr} \to \mathit{Bool}\\ &\mathit{isPos}\ (\mathit{Val}\ n) \quad = n > 0\\ &\mathit{isPos}\ (\mathit{Add}\ x\ y) = \mathit{isPos}\ x \wedge \mathit{isPos}\ y\end{aligned}$$

The approach we use to calculate type checking functions 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:

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

$$\textbf{data}\ \mathit{Value} = \mathit{I}\ \mathit{Int} \mid \mathit{B}\ \mathit{Bool} \mid \mathit{Error}$$

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:

$$\begin{aligned}&\mathit{eval} :: \mathit{Expr} \to \mathit{Value}\\ &\mathit{eval}\ (\mathit{Val}\ v) \quad = v\\ &\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{add}\ (\mathit{eval}\ x)\ (\mathit{eval}\ y)\\ &\mathit{eval}\ (\mathit{If}\ x\ y\ z) = \mathit{cond}\ (\mathit{eval}\ x)\ (\mathit{eval}\ y)\ (\mathit{eval}\ z)\end{aligned}$$

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

$$\begin{aligned}&\mathit{add} :: \mathit{Value} \to \mathit{Value} \to \mathit{Value}\\ &\mathit{add}\ (\mathit{I}\ n)\ (\mathit{I}\ m) = \mathit{I}\ (n + m)\\ &\mathit{add}\ \_ \quad \_ \quad = \mathit{Error}\end{aligned}$$

$$\begin{aligned}&\mathit{cond} :: \mathit{Value} \to \mathit{Value} \to \mathit{Value} \to \mathit{Value}\\ &\mathit{cond}\ (\mathit{B}\ b)\ v\ w = \textbf{if}\ b\ \textbf{then}\ v\ \textbf{else}\ w\\ &\mathit{cond}\ \_ \quad \_\ \_ = \mathit{Error}\end{aligned}$$

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.

Badly-formed expressions could also be handled in other ways, such as by replacing the explicit error value by the

Maybe monad and defining $eval :: Expr \to Maybe\ Value$, or by 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 [2].

4 Type Checking

The evaluation function *eval* 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 type checking function $texp :: Expr \to Type$ 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 captures the basic idea, it is also too strong. To see why, consider the following expression, written in regular Haskell syntax:

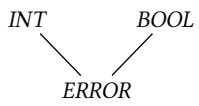
$$\textbf{if}\ True\ \textbf{then}\ 1\ \textbf{else}\ False$$

Evaluating this expression gives the value 1, an integer. However, for the type checker to determine this, it would need to use the value of the condition 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 instance 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:



We can view this as an *information ordering*, where $t \leqslant t'$ means 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 $\geqslant$, 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 desired 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. Our calculational approach 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 function *texp* and then proving it is correct, we can also use the specification

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

as the basis for *calculating* the definition of *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, $tval\ (eval\ e)$, and seek to strengthen it into something of the form $texp\ e$ by defining a suitable clause for the function in this case. Generalising from Section 2, each calculation step is either an equality (=) or a reverse inclusion ($\geqslant$).

Case: $Val\ v$

```
  tval (eval (Val v))
=   { applying eval }
  tval v
=   { define: texp (Val v) = tval v }
  texp (Val v)
```

The last step introduces a clause for *texp* so that the final term of the calculation is of the desired form that matches the left-hand side of the specification.

Case: $Add\ x\ y$

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

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:

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

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:

$$
\begin{array}{c}
\mathit{tval}\ (\mathit{add}\ (\mathit{eval}\ x)\ (\mathit{eval}\ y)) \\
\geqslant \\
\mathit{add}'\ (\mathit{tval}\ (\mathit{eval}\ x))\ (\mathit{tval}\ (\mathit{eval}\ y))
\end{array}
$$

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:

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

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

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

To discharge the assumptions made in the above calculation, it remains to define the operation $\mathit{add}' :: \mathit{Type} \to \mathit{Type} \to \mathit{Type}$ on types, 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, $\mathit{add}'\ (\mathit{tval}\ v)\ (\mathit{tval}\ w)$:

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

In summary, we have established an equality between the two sides, which is stronger than the required inequality, and have therefore calculated the following definition:

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

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

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

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.

Case: $\mathit{If}\ x\ y\ z$

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:

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

```
        cond′ (texp x) (texp y) (texp z) }
  texp (If x y z)
```

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:

```
  tval (cond c v w)
=   { applying cond }
  tval (case c of
    B b → if b then v else w
    _   → Error)
=   { distribution, tval is strict }
  case c of
    B b → if b then tval v else tval w
    _   → tval Error
=   { applying tval }
  case c of
    B b → if b then tval v else tval w
    _   → ERROR
⩾   { lemma below, case is monotonic }
  case c of
    B b → if tval v == tval w then tval v else ERROR
    _   → ERROR
=   { unapplying tval }
  case (tval c) of
    BOOL → if tval v == tval w then tval v else ERROR
    _    →      ERROR
=   { define:
      cond′ BOOL t  t′ = if t == t′ then t else ERROR
      cond′ _    _ _ = ERROR }
  cond′ (tval c) (tval v) (tval w)
```

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

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

We can prove this lemma by case analysis on $t == t'$. If this value is true, the lemma simplifies to (**if** b **then** t **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 (**if** b **then** t **else** t') $\geqslant$ *ERROR*, which is true because *ERROR* is the smallest type by definition.

In summary, we have calculated the following definition for *cond′*, 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 Using Fold and Fusion

An important aspect of our development above was the use of operations *add* and *cond* on values, and 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 [12], which takes the operation to apply for each form of expression as an additional 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 define evaluation and type checking in a more concise manner by 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 [12], which allows a function h that satisfies suitable distributivity, or *homomorphism*, properties to be fused with a fold over one collection of operations, *val*, *add* and *cond*, to give a fold over another collection, *val′*, *add′* and *cond′*. In our setting, we use a variant of fusion where equality (=) is replaced by a pre-order (⩾), under which the operations *add′*

and *cond′* are required to be monotonic:

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

This fusion property can be proved by straightforward induction on expressions. Using the above ideas, it now becomes evident that the correctness of type checking is just an application of fold fusion. In particular, the specification

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

can now be expanded to give:

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

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 definitions for *add′* and *cond′* as shown in the previous section.

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

7 Using Constraints

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

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

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, namely:

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

However, there is another approach to calculating *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 the four possible cases of this argument, and simplifying the homomorphism property for conditionals in each of these cases. In each case, we seek to obtain a simpler property that implies the homomorphism property. Hence, as in Section 2, each calculation step is either a logical equivalence or a reverse implication.

Case: $x = B\ True$

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

Case: $x = B\ False$

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

Case: $x = I\ n$

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

Case: $x = Error$

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

The above reasoning shows that the following four constraints for the operation *cond′* are together sufficient to

satisfy the homomorphism property for conditionals:

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

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 on types can be defined by the following three equations,

$$\begin{aligned} &cond'\ BOOL\ \ t\ t' = t \sqcap t' \\ &cond'\ INT\ \ \ \ t\ t' = ERROR \\ &cond'\ ERROR\ t\ t' = ERROR \end{aligned}$$

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

$$\begin{aligned} &cond'\ BOOL\ t\ \ t' = t \sqcap t' \\ &cond'\ \_\ \ \ \ \_\ \_ = ERROR \end{aligned}$$

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:

$$\begin{aligned} &(\sqcap) :: Type \to Type \to Type \\ &t \sqcap t' = \textbf{if}\ t == t'\ \textbf{then}\ t\ \textbf{else}\ ERROR \end{aligned}$$

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:

$$\begin{aligned} &cond' :: Type \to Type \to Type \to Type \\ &cond'\ BOOL\ t\ \ t' = \textbf{if}\ t == t'\ \textbf{then}\ t\ \textbf{else}\ ERROR \\ &cond'\ \_\ \ \ \ \_\ \_ = ERROR \end{aligned}$$

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 $y = I\ m$

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

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

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

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

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

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

$$\begin{aligned} &add' :: Type \to Type \to Type \\ &add'\ INT\ INT = INT \\ &add'\ \_\ \ \ \_\ \ \ = ERROR \end{aligned}$$

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 gives flexibility to handle more complex types in later sections.

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:

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

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:

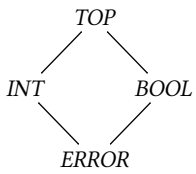
$$cond'\ s\ t\ t' \leqslant t \qquad \text{if } s \leqslant BOOL \tag{2}$$

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 $s = BOOL$. The converse implication from (1) to (2) can be verified as follows:

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

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

In the next step, we transform (2) to remove the need for $s \leqslant BOOL$ 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*:



The new element *TOP* is only needed for the purpose of calculation. Indeed, from our specification $texp\ e \leqslant tval\ (eval\ e)$, 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 $texp\ e = TOP$ as this would violate the specification. Using this new element, we can now write constraint (2) in the following equivalent form:

$$cond'\ s\ t\ t' \quad \leqslant \quad \textbf{if } s \leqslant BOOL \textbf{ then } t \textbf{ else } TOP \qquad (3)$$

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

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

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

$$cond'\ s\ t\ t' \quad \leqslant \quad (s \leqslant BOOL) \Rrightarrow t \qquad (4)$$

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:

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

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

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

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:

$$\begin{array}{lcl} add'\ t\ t' & \leqslant & (t \leqslant INT \land t' \leqslant INT) \Rrightarrow INT \\ add'\ t\ t' & \leqslant & (t \leqslant BOOL \lor t' \leqslant BOOL) \Rrightarrow ERROR \end{array}$$

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

$$\begin{array}{l} add' :: Type \to Type \to Type \\ add'\ t\ t' = ((t \leqslant INT \land t' \leqslant INT) \Rrightarrow INT) \\ \qquad\qquad \sqcap\ ((t \leqslant BOOL \lor t' \leqslant BOOL) \Rrightarrow ERROR) \end{array}$$

That is, if both arguments 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 *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:

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

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

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

In turn, using $\Rrightarrow$-subsume we can simplify the second half:

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

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

$$cond'\ s\ t\ t' = ((s \leqslant BOOL) \Rrightarrow t \sqcap t') \sqcap ((s \leqslant INT) \Rrightarrow ERROR)$$

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 a 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 the definition of *add′*.

9 Exception Handling

As a more sophisticated example of our approach, 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, *Throw* represents an exception that has been thrown, while *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 *Catch*:

```
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 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 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 an error in the first argument to be propagated. Conditionals are treated 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
```

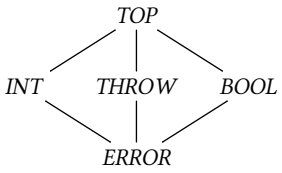
Finally, the new operation *catch* handles an exception thrown in the first argument by returning the second, and otherwise simply returns the value of the first argument:

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

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:

```
data Type = INT | BOOL | THROW | ERROR | TOP
tval :: Value → Type
tval (I _)  = INT
tval (B _)  = BOOL
tval Throw  = THROW
tval Error  = ERROR
```

The partial ordering on types is extended as shown below:



As in Section 8, we include *TOP* as the greatest element 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 $texp :: Expr \to Type$ that returns the type of an expression, with the desired behaviour being captured in the same manner as previously:

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

To calculate *texp*, we use the algebraic approach from Section 6, and assume it is defined by folding suitable operations on types, which themselves remain to be defined:

$$texp = folde\ val'\ add'\ cond'\ catch'$$

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

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

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 $val'\ x \leqslant tval\ (id\ x)$ simplifies immediately to $val'\ x \leqslant tval\ x$, and hence the optimal definition for *val*′ is to simply define:

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

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

$$add'\ t\ t' \quad \leqslant \quad (t \leqslant INT \land t' \leqslant INT) \Rrightarrow INT$$

The remaining cases lead to further constraints:

Case: $x = Throw$

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

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

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

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

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

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:

$$\begin{aligned} add'\ t\ t' \ &\leqslant\ (t \leqslant INT \land t' \leqslant INT) &&\Rrightarrow INT \\ add'\ t\ t' \ &\leqslant\ (t \leqslant THROW) &&\Rrightarrow THROW \\ add'\ t\ t' \ &\leqslant\ (t \leqslant INT \land t' \leqslant THROW) &&\Rrightarrow THROW \\ add'\ t\ t' \ &\leqslant \\ &\quad (t \leqslant BOOL \lor (t \leqslant INT \land t' \leqslant BOOL)) &&\Rrightarrow ERROR \end{aligned}$$

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:

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

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

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

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

Case: $x = Throw$

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

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

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

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:

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

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: $x = Throw$

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

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

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

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 obtain the optimal definition for $catch'$:

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

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.

10 Checked Exceptions

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 how exceptions are treated in conditionals. 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 conditional expression

$$\textbf{if } \mathit{True} \textbf{ then } 1 \textbf{ else } \mathit{Throw}$$

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:

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

Using this definition, the example expression has type

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

In contrast, in Haskell the similar expression

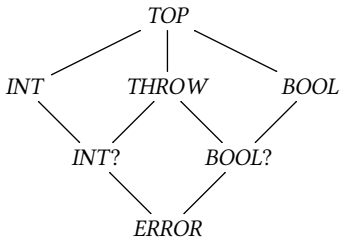
$$\textbf{if } \mathit{True} \textbf{ then } 1 \textbf{ else } \mathit{throw\ SomeException}$$

type checks fine as an integer. In Haskell, this is achieved *throw* having a polymorphic type, $\mathit{Exception}\ e \Rightarrow e \to a$. However, this approach might be viewed as unsatisfactory, as the 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 more informatively, 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 *INT*? that represents a value that could be either an integer or an exception, and a new type *BOOL*? that represents a value that could be either a logical value or an exception:

$$\begin{aligned} \textbf{data } \mathit{Type} = \mathit{INT} \mid \mathit{INT}? \mid \mathit{BOOL} \mid \mathit{BOOL}? \\ \mid \mathit{THROW} \mid \mathit{ERROR} \mid \mathit{TOP} \end{aligned}$$

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



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 $\mathit{tval} :: \mathit{Value} \to \mathit{Type}$ 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 optimal. Hence, the definition of the type checker $\mathit{texp} :: \mathit{Expr} \to \mathit{Type}$ 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 form of types. Crucially, however, the type checker can now give more informative results. For example, type checking **if** *True* **then** 1 **else** *Throw* now gives:

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

That is, the type checker now indicates that the expression could either produce an integer or raise an exception, whereas previously it simply gave a type error. As another example, consider *catch* (**if** *True* **then** 1 **else** *Throw*) 2. As shown above, the conditional expression has type *INT*?, hence the whole expression has type:

$$\mathit{catch}'\ \mathit{INT}?\ \mathit{INT} \;=\; \mathit{INT} \sqcap \mathit{INT} \;=\; \mathit{INT}$$

In this manner, the type checker recognises that catch handles the possibility of an exception being raised in the conditional, ensuring that an integer is always returned.

11 Related Work

The idea of calculating type checkers from (in)equational specifications is inspired by the compiler calculation methodology of Bahr and Hutton [2, 3]. However, calculating type checkers has required further refinements: using an ordered version of fold fusion to calculate inequational constraints, and a means to obtain optimal solutions for these constraints in a compositional manner.

A type checker is a special case of the notion of an *abstract interpreter* [5]. The soundness of an abstract interpreter f_a with respect to a concrete interpreter f_c is typically formulated as 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 both the concretisation function γ and the second inequality. While we only consider simple languages here, there is reason for optimism that such a simplification may generalise to higher-order and

non-terminating languages. In particular, Bahr and Hutton [2] have shown that the compiler calculation technique of Meijer [11] based on adjunctions, which generalise 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* [7, 9], a feature popularised in Java [8]. Exceptions and exception handling are a special case of *algebraic effects* and *effect handling* [15]. Type and effect systems have been proposed to check for the presence and proper handling of effects [4]. This suggests further work to explore how type and effects systems for algebraic effects can be calculated using the kind of techniques presented in this article.

12 Conclusion and Further Work

We showed how the calculational approach to program construction can be applied to the design of type-checking functions. In particular, we identified algebraic properties of both the evaluation semantics and the type language that allowed us to derive correct-by-construction type checkers in a principled and compositional manner.

There are a number of interesting topics for further work in this area. First of all, it would be useful to be able to calculate the language of types at the same time as the type checker itself, in a similar manner to how Bahr and Hutton [2] calculate the language of code at the same time as a compiler. Secondly, it is important to consider how the approach can be scaled to more sophisticated source language and type system features, such as non-termination, higher-order languages, computational effects, type constructors, and polymorphism. And finally, it would be interesting to calculate typing rules as opposed to type checkers, and we already have some promising results in this direction.

Acknowledgements

We thank the referees for their useful comments and suggestions. 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*.

References

[1] Roland Backhouse. 2003. *Program Construction: Calculating Implementations from Specifications*. John Wiley and Sons, Inc.
[2] Patrick Bahr and Graham Hutton. 2015. Calculating Correct Compilers. *Journal of Functional Programming* 25 (2015).
[3] Patrick Bahr and Graham Hutton. 2020. Calculating Correct Compilers II: Return of the Register Machines. *Journal of Functional Programming* 30 (2020).
[4] Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science*. Springer Berlin Heidelberg.
[5] Patrick Cousot. 1997. Types as Abstract Interpretations. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, Paris, France.
[6] Zac Garby, Patrick Bahr, and Graham Hutton. 2025. Haskell and Agda code for "The Calculated Typer". doi:10.5281/zenodo.16751640
[7] John B. Goodenough. 1975. Exception Handling: Issues and a Proposed Notation. *Commun. ACM* 18, 12 (1975).
[8] James Gosling, Bill Joy, and Guy L. Steele. 1996. *The Java Language Specification*. Addison-Wesley Longman Publishing.
[9] Barbara H. Liskov and Alan Snyder. 1979. Exception Handling in CLU. *IEEE Transactions on Software Engineering* 5, 6 (1979).
[10] Simon Marlow et al. 2010. Haskell 2010 Language Report. *Available online at https://www.haskell.org* (2010).
[11] Erik Meijer. 1992. *Calculating Compilers*. PhD Thesis. Katholieke Universiteit Nijmegen.
[12] 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*.
[13] Flemming Nielson, Hanne Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer.
[14] Benjamin C. Pierce. 2002. *Types and Programming Languages*. The MIT Press.
[15] 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.

Received 2025-06-06; accepted 2025-07-17