Sound-By-Construction Type Systems (Functional Pearl)

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

Type systems for programming languages are usually designed by hand, with the aim of satisfying a type soundness property that guarantees well-typed programs cannot go wrong. In this article, we show how standard techniques for *proving* type soundness can be used in reverse to systematically *derive* type systems that are sound by construction. We introduce and illustrate our methodology with a series of practical examples, including a typed lambda calculus with conditionals and checked exceptions.

1 Introduction

A type system is a set of rules that specify how types can be assigned to each component of a program in a particular language [Pierce 2002]. In this article, we focus on the process of *designing* type systems. The traditional approach is to design type systems by hand, by first devising a set of typing rules that formalise how types are assigned to program terms, and then proving a type soundness property that guarantees 'well-typed programs cannot go wrong' [Milner 1978]. In practice, this is usually an iterative process that requires quite a bit of trial and error, rather than simply writing down a set of rules and then proving soundness.

Here we take a different approach. In particular, we show how standard techniques for *proving* type soundness can be used in reverse to systematically *derive* type systems that are sound. The starting point for our approach is a semantics for the language being considered, expressed as a big-step evaluation relation. We then formulate a suitable type soundness property, which formalises the idea that well-typed programs are guaranteed to evaluate successfully. And finally, we calculate a set of typing rules by 'solving' the type soundness property using algebraic reasoning techniques, in a similar manner to how equations are solved in mathematics.

The calculational approach to type system design has a number of benefits. First of all, type systems produced in this manner are *sound by construction*, and hence require no separate proofs of soundness after they have been produced. Secondly, the approach provides a principled way to *discover* typing rules, and to explore alternative design choices during and after the calculation process. And finally, the approach lends itself naturally to mechanical *formalisation*, allowing proof assistants to be used as interactive tools for developing and certifying the calculations.

The aim of this article is to present a basic methodology for calculating type systems from soundness properties, and establish its feasibility. Our presentation is example-driven, and is structured around a series of languages of increasing complexity:

- We begin with a simple expression language with conditionals, show how to calculate a type system for this language, and explore alternative design choices (section 2);
- We then extend the language with support for exception handling, and show how to calculate a type system that supports the idea of checked exceptions (section 3);
- And finally, as more sophisticated examples, we calculate a type system for the lambda calculus (section 4), and an extension with conditionals and exceptions (section 5).

Despite the fact that type systems and program calculation are both long-standing research topics, to the best of our knowledge these ideas have not been combined before. We consider minimal languages with particular features of interest, as this allows us to focus on core principles

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

and present many of the calculations in detail: our emphasis is on the calculation *process*, the journey, rather than solely on the resulting type system, the destination. We restrict our attention to languages with deterministic big-step semantics and type soundness properties that include strong normalisation, and discuss possible generalisations in further work.

The article is aimed at readers with some basic experience of formal semantics and reasoning, but does not require specialist knowledge about type systems. All of the examples have also been formalised in Agda, and the code is freely available as supplementary material.

2 Conditional Language

Consider a simple expression language built up from basic values using addition and conditional operations, where a value is either an integer $n \in \mathbb{Z}$ or a logical value $b \in \{\mathsf{false}, \mathsf{true}\}$:

$$e ::= v \mid \mathsf{add}\, e\, e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \qquad\qquad v ::= n \mid b$$

We define the semantics of the language using an evaluation relation $e \Downarrow v$ that specifies when an expression e can evaluate to a value v, which is given by the following collection of rules:

$$\frac{}{v \Downarrow v} \qquad \frac{e \Downarrow n \qquad e' \Downarrow n'}{\mathsf{add}\, e\, e' \Downarrow n + n'} \qquad \frac{e \Downarrow \mathsf{true} \qquad e_1 \Downarrow v}{\mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \Downarrow v} \qquad \frac{e \Downarrow \mathsf{false} \qquad e_2 \Downarrow v}{\mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \Downarrow v}$$

By convention, we use metavariables e to range over expressions, v over values, n over integers, and b over logical values. For example, the first rule above has the implicit side condition that v is a value, and the second that n and n' are integers.

It is straightforward to show that evaluation is deterministic, i.e. every expression evaluates to at most one value. However, evaluation can also fail, in particular when attempting to add values that are not integers, or make a conditional choice on a value that is not true or false. Using a relational semantics naturally captures the possibility of failure, as relations can be partial. In contrast, a functional semantics typically requires an explicit failure value to ensure totality, which complicates the semantics and reasoning about it. With the relational approach, failure of an expression e to evaluate simply means there is no value v such that $e \Downarrow v$.

2.1 Type Soundness

A common method for ensuring successful evaluation is to introduce a type system that guarantees this. We first define a simple language of types comprising integers and logical values,

$$t ::= \mathsf{Int} \mid \mathsf{Bool}$$

together with a semantic function $[\![-]\!]$ that maps each type to the set of values it represents:

$$[\![\mathsf{Int}]\!] = \mathbb{Z} \qquad [\![\mathsf{Bool}]\!] = \{\mathsf{false}, \mathsf{true}\}$$

Suppose now that we wish to define a typing relation $\vdash e : t$ that specifies when an expression e can have type t. The desired behaviour is captured by the following *type soundness* property:

$$\frac{\vdash e : t}{\exists v.\ e \Downarrow v \wedge v \in [\![t]\!]} \tag{1}$$

This property states that if an expression has a particular type, then the expression can always evaluate to a value of this type. In combination with the evaluation relation being deterministic, type soundness ensures that 'well-typed expressions cannot go wrong' [Milner 1978], i.e. they are guaranteed to successfully evaluate to a value of the given type.

Note that the above specification gives flexibility in how the typing relation $\vdash$ is implemented, as there may be many possible definitions that satisfy the type soundness property. Indeed, the empty typing relation is trivially sound, and so is the relation given by simply defining $\vdash e : t$ to mean $\exists v.\ e \Downarrow v \wedge v \in [\![t]\!]$. Our calculational approach naturally avoids such trivial solutions, and provides a systematic means for designing typing rules that satisfy the above specification.

2.2 Soundness by Construction

Now that we have formulated the type soundness property, the traditional approach at this point is to manually define the typing relation $\vdash e : t$ using a collection of rules, and then prove that the relation defined by these rules satisfies the soundness property [Pierce 2002].

However, rather than first defining the typing relation and then separately proving that it is sound, we can also use soundness as the basis for directly *calculating* the definition of the relation. That is, we can seek to derive the typing relation by solving the soundness property for this relation, in much the same way as we solve equations in mathematics. A type system produced in this manner is *sound by construction*, and hence requires no separate proof of soundness.

The starting point for our methodology to achieve the above is to define a *semantic* typing relation that exactly captures the desired soundness property:

$$\models e : t \qquad \stackrel{\text{def}}{\iff} \qquad \exists v.\ e \Downarrow v \wedge v \in [\![t]\!]$$

Using this definition, the original type soundness property (1) can now be formulated as:

$$\frac{\vdash e : t}{\models e : t} \tag{2}$$

To derive rules for the typing relation $\vdash$ that satisfy this soundness property by construction, we aim to calculate properties of the semantic relation $\models$ that have the form

$$\frac{\models e_1 : t_1 \quad \dots \quad \models e_n : t_n \qquad P}{\models e : t} \tag{3}$$

where P is any additional premise that does not refer to $\models$, which can be used to impose side conditions such as a value being in a given set, or an expression evaluating to a given value. We refer to a property of the above form as a *semantic typing rule* or simply as a *rule* for $\models$.

Once we have calculated these rules for $\models$, we then *inductively define* the typing relation $\vdash$ by rules that have precisely the same form as those for $\models$, but with $\models$ replaced by $\vdash$:

$$\frac{\vdash e_1 : t_1 \quad \dots \quad \vdash e_n : t_n \qquad P}{\vdash e : t} \tag{4}$$

Because the relation $\models$ occurs only positively in the premises of the rules of form (3), we obtain a valid inductive definition for $\vdash$. Moreover, both relations $\models$ and $\vdash$ satisfy the same collection of properties, and because $\vdash$ is defined inductively, it is the smallest such relation. Therefore, we have $\vdash \subseteq \models$, which is precisely the point-free form of the type soundness property (2).

More formally, the above reasoning is justified by the Knaster-Tarski theorem [Tarski 1955]. Define a *typing relation* $\mathcal{R}$ to be a relation between expressions and types. We then translate each rule of form (3) into a function F on typing relations as follows:

$$\frac{\models e_1 : t_1 \quad \dots \quad \models e_n : t_n \qquad P}{\models e : t} \quad \longmapsto \quad F(\mathcal{R}) = \{(e, t) \mid (e_1, t_1) \in \mathcal{R} \wedge \cdots \wedge (e_n, t_n) \in \mathcal{R} \wedge P\}$$

For example, a rule for the addition of integer expressions would be translated by:

$$\frac{\models e_1 : \mathsf{Int} \qquad \models e_2 : \mathsf{Int}}{\models \mathsf{add}\, e_1\, e_2 : \mathsf{Int}} \quad \longmapsto \quad F(\mathcal{R}) = \{(\mathsf{add}\, e_1\, e_2, \mathsf{Int}) \mid (e_1, \mathsf{Int}) \in \mathcal{R} \wedge (e_2, \mathsf{Int}) \in \mathcal{R}\}$$

Let $\mathcal{F}$ be the pointwise union of such functions for a given set of rules. Crucially, $\mathcal{F}$ is monotonic with respect to the inclusion ordering $\subseteq$ on relations, because the rules only refer positively to $\models$ in the premises. A typing relation $\mathcal{R}$ *satisfies the rules* iff $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{R}$, which means that $\mathcal{R}$ is a pre-fixpoint of $\mathcal{F}$. Because $\models$ satisfies the rules, we have that $\models$ is a pre-fixpoint of $\mathcal{F}$. Moreover, by Knaster-Tarski, we know that $\mathcal{F}$ has a unique least pre-fixpoint, which we denote by $\vdash$. Therefore, $\vdash$ is well-defined, and because $\vdash$ is the least pre-fixpoint, we have $\vdash \subseteq \models$. In the remainder of the article, we show how the above ideas can be used to obtain sound type systems by calculation.

2.3 Calculating a Type System

We now proceed to calculate a semantic typing rule for each rule of the evaluation relation as follows: we start with a term of the form $\models e : t$, where e is the expression being evaluated, and seek to strengthen it into a conjunction of terms of the form $\models e_i : t_i$ together with an optional additional premise P that does not refer to $\models$, thus matching the premise of the required form (3) for semantic typing rules. First of all, the case for values simplifies to a premise that on its own establishes the desired result, i.e. no assumptions involving $\models$ itself are required.

Case: $\dfrac{}{v \Downarrow v}$

$$
\begin{array}{ll}
& \models v : t \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& \exists v'.\, v \Downarrow v' \wedge v' \in [\![t]\!] \\
\Leftrightarrow & \{ \text{ definition of } \Downarrow \} \\
& \exists v'.\, v' = v \wedge v' \in [\![t]\!] \\
\Leftrightarrow & \{ \text{ substitute } v' = v \} \\
& v \in [\![t]\!]
\end{array}
$$

That is, we have calculated the semantic typing rule

$$\frac{v \in [\![t]\!]}{\models v : t}$$

which we can then instantiate using the definition of the semantic function $[\![-]\!]$ to give two semantic typing rules, one for each of the basic types Int and Bool:

$$\frac{n \in \mathbb{Z}}{\models n : \mathsf{Int}} \qquad\qquad \frac{b \in \{\mathsf{false}, \mathsf{true}\}}{\models b : \mathsf{Bool}}$$

Note that the calculation above actually yields properties that are equivalences, but for the purposes of our methodology we only require that they are implications, namely rules of the form (3).

The case for addition proceeds similarly, by first using the definitions of $\models$ and $\Downarrow$, and then simplifying the resulting term. Using the semantics of the Int type and separating the two remaining quantifiers then allows the term to be rewritten into the required conjunctive form.

Case: $\dfrac{e \Downarrow n \qquad e' \Downarrow n'}{\mathsf{add}\; e\; e' \Downarrow n + n'}$

$$
\begin{array}{ll}
& \models \mathsf{add}\; e\; e' : t \\
\Leftrightarrow & \{ \text{ definition of } \models \}
\end{array}
$$

$$
\begin{array}{ll}
 & \exists v.\ \mathsf{add}\ e\ e' \Downarrow v \land v \in [\![t]\!] \\
\Leftrightarrow & \{\ \text{definition of } \Downarrow\ \} \\
 & \exists v.\ \exists n, n'.\ e \Downarrow n \land e' \Downarrow n' \land v = n + n' \land v \in [\![t]\!] \\
\Leftrightarrow & \{\ \text{substitute } v = n + n'\ \} \\
 & \exists n, n'.\ e \Downarrow n \land e' \Downarrow n' \land n + n' \in [\![t]\!] \\
\Leftrightarrow & \left\{\ n, n' \in \mathbb{Z} \text{ by convention, definition of } [\![-]\!]\ \right\} \\
 & \exists n, n'.\ e \Downarrow n \land n \in [\![\mathsf{Int}]\!] \land e' \Downarrow n' \land n' \in [\![\mathsf{Int}]\!] \land t = \mathsf{Int} \\
\Leftrightarrow & \{\ \text{separate quantifiers}\ \} \\
 & (\exists n.\ e \Downarrow n \land n \in [\![\mathsf{Int}]\!]) \land (\exists n'.\ e' \Downarrow n' \land n' \in [\![\mathsf{Int}]\!]) \land t = \mathsf{Int} \\
\Leftrightarrow & \{\ \text{definition of } \models\ \} \\
 & \models e : \mathsf{Int} \land \models e' : \mathsf{Int} \land t = \mathsf{Int}
\end{array}
$$

That is, we have calculated the semantic typing rule

$$
\frac{\models e : \mathsf{Int} \qquad \models e' : \mathsf{Int} \qquad t = \mathsf{Int}}{\models \mathsf{add}\ e\ e' : t}
$$

which can then be written in a simpler but equivalent form as:

$$
\frac{\models e : \mathsf{Int} \qquad \models e' : \mathsf{Int}}{\models \mathsf{add}\ e\ e' : \mathsf{Int}}
$$

We often apply similar simplifications to the calculated semantic typing rules.

Finally, for conditionals there are two cases, depending on whether the condition is true or false. The key step is once again a simple quantifier manipulation to obtain terms of the required form.

Case: $\dfrac{e \Downarrow \mathsf{true} \qquad e_1 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v}$

$$
\begin{array}{ll}
 & \models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t \\
\Leftrightarrow & \{\ \text{definition of } \models\ \} \\
 & \exists v.\ \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v \land v \in [\![t]\!] \\
\Leftarrow & \{\ \text{definition of } \Downarrow\ \} \\
 & \exists v.\ e \Downarrow \mathsf{true} \land e_1 \Downarrow v \land v \in [\![t]\!] \\
\Leftrightarrow & \{\ \text{move quantifier inside}\ \} \\
 & e \Downarrow \mathsf{true} \land (\exists v.\ e_1 \Downarrow v \land v \in [\![t]\!]) \\
\Leftrightarrow & \{\ \text{definition of } \models\ \} \\
 & e \Downarrow \mathsf{true} \land \models e_1 : t
\end{array}
$$

Case: $\dfrac{e \Downarrow \mathsf{false} \qquad e_2 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v}$

$$
\begin{array}{ll}
 & \models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t \\
\Leftrightarrow & \{\ \text{definition of } \models\ \} \\
 & \exists v.\ \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v \land v \in [\![t]\!] \\
\Leftarrow & \{\ \text{definition of } \Downarrow\ \} \\
 & \exists v.\ e \Downarrow \mathsf{false} \land e_2 \Downarrow v \land v \in [\![t]\!] \\
\Leftrightarrow & \{\ \text{move quantifier inside}\ \} \\
 & e \Downarrow \mathsf{false} \land (\exists v.\ e_2 \Downarrow v \land v \in [\![t]\!]) \\
\Leftrightarrow & \{\ \text{definition of } \models\ \} \\
 & e \Downarrow \mathsf{false} \land \models e_2 : t
\end{array}
$$

That is, we have calculated the following semantic typing rules:

$$
\frac{e \Downarrow \mathsf{true} \qquad \models e_1 : t}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}
\qquad\qquad
\frac{e \Downarrow \mathsf{false} \qquad \models e_2 : t}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}
$$

Given all the calculated semantic typing rules for $\models$, we can now simply replace $\models$ by $\vdash$ in each rule and thereby obtain an inductive definition for a type system that is guaranteed to be sound by construction. The resulting type system is given in Figure 1.

$$\frac{n \in \mathbb{Z}}{\vdash n : \mathsf{Int}} \qquad \frac{b \in \{\mathsf{false}, \mathsf{true}\}}{\vdash b : \mathsf{Bool}} \qquad \frac{\vdash e : \mathsf{Int} \quad \vdash e' : \mathsf{Int}}{\vdash \mathsf{add}\, e\, e' : \mathsf{Int}} \qquad \frac{e \Downarrow \mathsf{true} \quad \vdash e_1 : t}{\vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t} \qquad \frac{e \Downarrow \mathsf{false} \quad \vdash e_2 : t}{\vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t}$$

Fig. 1. Derived type system for the conditional language.

The derived system is also as general as possible with respect to the soundness property from which it was obtained. In particular, every step in the calculation of rules for $\models$ that give rise to the system is an equivalence, except for the two cases for conditionals, which have one reverse implication step when the condition is either true or false. In this manner, we are not 'losing information' in the calculations by unnecessarily strengthening the term being manipulated, and hence obtain a type system that is maximally general. Indeed, for this simple language, type soundness is actually an equivalence, i.e. the derived type system is both sound and complete.

2.4 Other Typing Rules

While the derived typing rules in Figure 1 are as general as possible, they also go beyond what we might normally expect for a type system for this language, in two ways. First of all, the rules for conditionals depend on the value of the condition, i.e. whether it is true or false, rather than just depending on its type. And secondly, the rules allow the unused branch of a conditional to be ill-typed, as there are no premises for the unused branches in the typing rules.

However, we can use the rules that we derived for the semantic typing relation $\models$ to derive another valid rule that avoids these issues. In particular, we can start with the conjunction of the two derived rules for conditionals, and transform these into another rule. Note that when we perform logical reasoning in the premise of a rule, contravariance leads the overall logical implication to reverse direction, as it does in the third calculation step below.

$$\frac{e \Downarrow \mathsf{true} \qquad \models e_1 : t}{\models \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t} \;\wedge\; \frac{e \Downarrow \mathsf{false} \qquad \models e_2 : t'}{\models \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t'}$$

$\Rightarrow$ { instantiate $t' = t$ }

$$\frac{e \Downarrow \mathsf{true} \qquad \models e_1 : t}{\models \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t} \;\wedge\; \frac{e \Downarrow \mathsf{false} \qquad \models e_2 : t}{\models \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t}$$

$\Leftrightarrow$ { combine into one rule }

$$\frac{(e \Downarrow \mathsf{true} \wedge \models e_1 : t) \;\vee\; (e \Downarrow \mathsf{false} \wedge \models e_2 : t)}{\models \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t}$$

$\Rightarrow$ { bring evaluation terms together }

$$\frac{(e \Downarrow \mathsf{true} \vee e \Downarrow \mathsf{false}) \qquad \models e_1 : t \qquad \models e_2 : t}{\models \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t}$$

$\Leftrightarrow$ { definition of $[\![-]\!]$ }

$$\frac{(\exists v. e \Downarrow v \wedge v \in [\![\mathsf{Bool}]\!]) \qquad \models e_1 : t \qquad \models e_2 : t}{\models \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t}$$

$\Leftrightarrow$ { definition of $\models$ }

$$\frac{\models e : \mathsf{Bool} \qquad \models e_1 : t \qquad \models e_2 : t}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}$$

Hence, we may replace the earlier two syntactic typing rules for conditionals with the following single rule, which does not require evaluating the condition and ensures that both branches are well-typed. This is the kind of rule that is, for example, found in languages such as Haskell.

$$\frac{\vdash e : \mathsf{Bool} \qquad \vdash e_1 : t \qquad \vdash e_2 : t}{\vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}$$

As previously, our methodology ensures that the resulting typing system is sound by construction. Being able to use previously derived semantic typing rules to derive new syntactic rules in this manner is an important benefit of our calculational methodology.

As another example of this idea, we can combine the two original rules for conditionals into a single rule that still allows the two branches to have different types:

$$\frac{e \Downarrow \mathsf{true} \qquad \models e_1 : t}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t} \wedge \frac{e \Downarrow \mathsf{false} \qquad \models e_2 : t'}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t'}$$

$\Rightarrow$ { add extra premise to each rule }

$$\frac{e \Downarrow \mathsf{true} \qquad \models e_1 : t \qquad \models e_2 : t'}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t} \wedge \frac{e \Downarrow \mathsf{false} \qquad \models e_1 : t \qquad \models e_2 : t'}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t'}$$

$\Leftrightarrow$ { combine into one rule using a conditional }

$$\frac{e \Downarrow b \qquad \models e_1 : t \qquad \models e_2 : t'}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathit{if}\ b\ \mathit{then}\ t\ \mathit{else}\ t'}$$

$\Leftrightarrow$ { replace conditional by a function }

$$\frac{e \Downarrow b \qquad \models e_1 : T\ \mathsf{true} \qquad \models e_2 : T\ \mathsf{false}}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T\ b}$$

The function T introduced in the final step maps logical values to types. Hence, we may replace the earlier two syntactic typing rules for conditionals with the following single rule, which enforces that both branches are well-typed, but allows them to have different types:

$$\frac{e \Downarrow b \qquad \vdash e_1 : T\ \mathsf{true} \qquad \vdash e_2 : T\ \mathsf{false}}{\vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T\ b}$$

This rule can be seen as a weak form of the kind of typing rule that is found in dependently-typed languages such as Agda. The difference is that, in our setting, T is a meta-level function rather than a term in the source language. Once again, being able to derive such a typing rule in a principled manner is a key benefit of our calculational approach.

2.5 Reflection

We conclude this section with some reflective remarks on our new methodology.

Methodology. We have shown how a sound-by-construction type system for the conditional language can be developed using a four-step process, which is summarised below:

1. Define a deterministic big-step semantics for the language;
2. Define a semantic typing relation $\models$ that captures the desired soundness property;
3. Calculate rules of $\models$ that have the form (3) of a strictly positive inference rule;

4. Obtain a sound type system by interpreting the rules of $\models$ as inductive rules for $\vdash$.

In the third step, we calculated one semantic typing rule for each evaluation rule. Subsequently, the derived semantic typing rules can be subjected to further calculation to obtain rules of a preferable form or to explore alternative rules. For example, we simplified the rule for addition, and we conjunctively combined multiple rules for conditionals and then simplified the result.

Calculation process. The derived typing rules for the conditional language are standard, and in themselves unsurprising. But as noted in the introductory section, our interest is in the process by which type systems can be derived, rather than just the resulting systems. In particular, the rules were obtained by systematic calculation, driven by the desire to ensure that type soundness is satisfied by construction. Our methodology also provides a principled way to explore alternative design choices. For example, we showed how the original typing rules for conditionals can be combined to calculate two different kinds of typing rules that enforce well-typing of both conditional branches. We will see further examples of this idea later on.

Non-inductive reasoning. The calculation of rules for the semantic typing relation $\models$ from which the typing rules were derived did not require any form of induction. Rather, the calculations proceed by simply applying definitions, using logical properties, and manipulating quantifiers. Induction only plays a role once we have derived suitable rules for $\models$, after which we then inductively define the typing relation $\vdash$ as the least relation satisfying these rules. In this manner, we can focus on the essential, non-inductive parts of the reasoning, with the use of induction being built-in to our methodology rather than having to be manually applied.

Semantic type soundness. A popular approach to establishing type soundness is based on the syntactic properties of progress and preservation [Wright and Felleisen 1994]. Here we use the approach of semantic type soundness, based on a denotational interpretation of types, which has seen a recent resurgence [Timany et al. 2024]. In addition to avoiding the need for inductive reasoning, as noted above, this approach enables more modular reasoning. For instance, the semantic approach allows the two cases for conditionals to be treated separately, whereas in the syntactic approach these usually need to be considered together, which leads to more cumbersome reasoning. As we will see, the semantic approach also scales elegantly to more sophisticated languages.

Mechanical formalisation. All the calculations in this section, and the rest of the article, have been formalised in the Agda proof assistant. The formalisation is straightforward, with the rules that define the typing relation $\vdash$ being implemented as an inductive family, and the soundness proof then obtained by induction over this family, where each case of the induction is precisely the corresponding semantic typing rule. The Agda code is freely available as supplementary material, and includes additional commentary on how the results are formalised.

3 Checked Exceptions

As a second example of our approach to calculating type systems, we now extend the conditional language from section 2 with support for throwing and catching an exception:

$$e ::= v \mid \mathsf{add}\; e\; e \mid \mathsf{if}\; e\; \mathsf{then}\; e\; \mathsf{else}\; e \mid \mathsf{try}\; e\; \mathsf{catch}\; e \qquad\qquad v ::= n \mid b \mid \mathsf{throw}$$

As previously, n is an integer and b is a logical value. Intuitively, the new value throw represents an exception that has been thrown, while an expression $\mathsf{try}\; e\; \mathsf{catch}\; e'$ behaves as the expression e unless it throws an exception, in which case it behaves as the *handler* expression e'. The semantics of the language is formally defined by the following evaluation rules:

$$\frac{}{v \Downarrow v} \qquad \frac{e \Downarrow n \qquad e' \Downarrow n'}{\mathsf{add}\; e\; e' \Downarrow n + n'} \qquad \frac{e \Downarrow \mathsf{true} \qquad e_1 \Downarrow v}{\mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2 \Downarrow v} \qquad \frac{e \Downarrow \mathsf{false} \qquad e_2 \Downarrow v}{\mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2 \Downarrow v}$$

$$\frac{e \Downarrow \mathsf{throw}}{\mathsf{add}\; e\; e' \Downarrow \mathsf{throw}} \qquad \frac{e \Downarrow n \qquad e' \Downarrow \mathsf{throw}}{\mathsf{add}\; e\; e' \Downarrow \mathsf{throw}} \qquad \frac{e \Downarrow \mathsf{throw}}{\mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2 \Downarrow \mathsf{throw}}$$

$$\frac{e \Downarrow \mathsf{throw} \qquad e' \Downarrow v}{\mathsf{try}\; e\; \mathsf{catch}\; e' \Downarrow v} \qquad \frac{e \Downarrow v \qquad v \neq \mathsf{throw}}{\mathsf{try}\; e\; \mathsf{catch}\; e' \Downarrow v}$$

The first four rules are the same as before. The five new rules that follow specify that addition propagates an exception thrown in either argument, conditionals propagate an exception thrown in their first argument, and try/catch handles an exception thrown in its first argument by returning the value of its second argument, and otherwise simply returns the value of the first.

We now extend the language of types to deal with exceptions. The approach we take is inspired by *checked exceptions* in Java [Gosling et al. 1996], where the signature for a method declares any exceptions that may be thrown in its body. To apply this idea in our setting, we extend the language of types with an operation ? that captures the possibility of an exception being thrown:

$$t ::= \mathsf{Int} \mid \mathsf{Bool} \mid t?$$

The intuition is that an expression of type $t?$ either evaluates to a value of type t, or results in an exception being thrown. This idea is formalised by extending the semantic function $[\![-]\!]$ that maps each type to the set of values it represents as follows:

$$[\![\mathsf{Int}]\!] = \mathbb{Z} \qquad [\![\mathsf{Bool}]\!] = \{\mathsf{false}, \mathsf{true}\} \qquad [\![t?]\!] = [\![t]\!] \cup \{\mathsf{throw}\}$$

Note that multiple uses of ? have no effect as it is semantically idempotent, i.e. $[\![(t?)?]\!] = [\![t?]\!]$. The semantic typing relation $\models$ is then defined in the same way as previously,

$$\models e : t \quad \stackrel{\text{def}}{\iff} \quad \exists v.\; e \Downarrow v \wedge v \in [\![t]\!]$$

and the desired type soundness property also remains the same:

$$\frac{\vdash e : t}{\models e : t}$$

3.1 Calculating a Type System

Prior to calculating typing rules for the language, we observe that because $[\![t?]\!]$ is by definition a superset of $[\![t]\!]$, the semantics for types naturally induces a *subtyping* relation $\leq$ defined as the least partial ordering satisfying $t \leq t?$. Then we have that $t \leq t'$ implies $[\![t]\!] \subseteq [\![t']\!]$, and thus we immediately obtain the following subtyping rule for the $\models$ relation:

$$\frac{\models e : t \qquad t \leq t'}{\models e : t'} \tag{5}$$

To derive rules for the typing relation $\vdash$ that satisfy the soundness property by construction, we use the methodology introduced in section 2 and first calculate rules for the semantic typing relation $\models$. In particular, for each rule in our evaluation semantics, we seek to strengthen the term $\models e : t$, where e is the expression being evaluated, into a conjunction of terms of the form $\models e_i : t_i$ together with an optional additional premise P that does not refer to $\models$.

Because the first four rules of the evaluation semantics are the same as previously, in these cases we derive the same rules for $\models$. In the case of values, instantiating the derived rule

$$\frac{v \in [\![t]\!]}{\models v : t}$$

using the semantic definition $[\![t?]\!] = [\![t]\!] \cup \{\mathsf{throw}\}$ for the new type $t?$ that deals with exceptions, and then simplifying, we obtain the following rule for the exceptional value throw:

$$\frac{}{\models \mathsf{throw} : t?}$$

Now we consider the new rules that specify that addition propagates an exception thrown in either argument. In each case, we use the fact that $\mathsf{throw} \in [\![t]\!]$ means that the type t must be of the form $t'?$, because ? is the only means of introducing the value throw into a semantic type.

Case: $\dfrac{e \Downarrow \mathsf{throw}}{\mathsf{add}\, e\, e' \Downarrow \mathsf{throw}}$

$$\begin{array}{ll}
& \models \mathsf{add}\, e\, e' : t \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& \exists v.\ \mathsf{add}\, e\, e' \Downarrow v \land v \in [\![t]\!] \\
\Leftarrow & \{\ v = \mathsf{throw}, \text{ definition of } \Downarrow \} \\
& e \Downarrow \mathsf{throw} \land \mathsf{throw} \in [\![t]\!] \\
\Leftrightarrow & \left\{ \text{ definition of } [\![-]\!] \right\} \\
& e \Downarrow \mathsf{throw} \land \exists t'.\ t = t'?
\end{array}$$

Case: $\dfrac{e \Downarrow n \qquad e' \Downarrow \mathsf{throw}}{\mathsf{add}\, e\, e' \Downarrow \mathsf{throw}}$

$$\begin{array}{ll}
& \models \mathsf{add}\, e\, e' : t \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& \exists v.\ \mathsf{add}\, e\, e' \Downarrow v \land v \in [\![t]\!] \\
\Leftarrow & \{\ v = \mathsf{throw}, \text{ definition of } \Downarrow \} \\
& \exists n.\ e \Downarrow n \land e' \Downarrow \mathsf{throw} \land \mathsf{throw} \in [\![t]\!] \\
\Leftrightarrow & \{ \text{ move quantifier inside } \} \\
& (\exists n.\ e \Downarrow n) \land e' \Downarrow \mathsf{throw} \land \mathsf{throw} \in [\![t]\!] \\
\Leftrightarrow & \left\{\ n \in \mathbb{Z} \text{ by convention, definition of } [\![-]\!] \right\} \\
& (\exists n.\ e \Downarrow n \land n \in [\![\mathsf{Int}]\!]) \land e' \Downarrow \mathsf{throw} \land \exists t'.\ t = t'? \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& \models e : \mathsf{Int} \land e' \Downarrow \mathsf{throw} \land \exists t'.\ t = t'?
\end{array}$$

That is, we have calculated the following semantic typing rules:

$$\frac{e \Downarrow \mathsf{throw} \qquad \exists t'.\ t = t'?}{\models \mathsf{add}\, e\, e' : t} \qquad\qquad \frac{\models e : \mathsf{Int} \qquad e' \Downarrow \mathsf{throw} \qquad \exists t'.\ t = t'?}{\models \mathsf{add}\, e\, e' : t}$$

By simple calculation, we can transform each of the above rules into an equivalent form that incorporates the equality premise $t = t'?$ into the conclusion:

$$\begin{array}{ll}
& \dfrac{e \Downarrow \mathsf{throw} \qquad \exists t'.\ t = t'?}{\models \mathsf{add}\, e\, e' : t} \\
\Leftrightarrow & \{ \text{ move quantifier outside } \} \\
& \forall t'.\ \dfrac{e \Downarrow \mathsf{throw} \qquad t = t'?}{\models \mathsf{add}\, e\, e' : t} \\
\Leftrightarrow & \{ \text{ substitute } t = t'? \ \} \\
& \forall t'.\ \dfrac{e \Downarrow \mathsf{throw}}{\models \mathsf{add}\, e\, e' : t'?}
\end{array}
\qquad
\begin{array}{ll}
& \dfrac{\models e : \mathsf{Int} \qquad e' \Downarrow \mathsf{throw} \qquad \exists t'.\ t = t'?}{\models \mathsf{add}\, e\, e' : t} \\
\Leftrightarrow & \{ \text{ move quantifier outside } \} \\
& \forall t'.\ \dfrac{\models e : \mathsf{Int} \qquad e' \Downarrow \mathsf{throw} \qquad t = t'?}{\models \mathsf{add}\, e\, e' : t} \\
\Leftrightarrow & \{ \text{ substitute } t = t'? \ \} \\
& \forall t'.\ \dfrac{\models e : \mathsf{Int} \qquad e' \Downarrow \mathsf{throw}}{\models \mathsf{add}\, e\, e' : t'?}
\end{array}$$

That is, we obtain the following simplified semantic typing rules:

$$\frac{e \Downarrow \mathsf{throw}}{\models \mathsf{add}\; e\; e' : t?} \qquad \frac{\models e : \mathsf{Int} \qquad e' \Downarrow \mathsf{throw}}{\models \mathsf{add}\; e\; e' : t?}$$

The case for conditionals propagating an exception thrown in their first argument proceeds in a similar manner to the case for addition, and results in the following semantic typing rule:

$$\frac{e \Downarrow \mathsf{throw}}{\models \mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2 : t?}$$

Finally, we consider the two rules that define the semantics for try/catch. The calculation for the first rule uses a simple quantifier manipulation to rewrite the term into the required form using $\models$, while the second rule uses the fact that evaluation is deterministic, i.e. every expression has at most one value, to split a quantifier into two parts:

Case: $\dfrac{e \Downarrow \mathsf{throw} \qquad e' \Downarrow v}{\mathsf{try}\; e\; \mathsf{catch}\; e' \Downarrow v}$

$$\begin{array}{cl}
& \models \mathsf{try}\; e\; \mathsf{catch}\; e' : t \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& \exists v.\ \mathsf{try}\; e\; \mathsf{catch}\; e' \Downarrow v \wedge v \in [\![t]\!] \\
\Leftarrow & \{ \text{ definition of } \Downarrow \} \\
& \exists v.\ e \Downarrow \mathsf{throw} \wedge e' \Downarrow v \wedge v \in [\![t]\!] \\
\Leftrightarrow & \{ \text{ move quantifier inside } \} \\
& e \Downarrow \mathsf{throw} \wedge (\exists v.\ e' \Downarrow v \wedge v \in [\![t]\!]) \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& e \Downarrow \mathsf{throw} \wedge \models e' : t
\end{array}$$

Case: $\dfrac{e \Downarrow v \qquad v \neq \mathsf{throw}}{\mathsf{try}\; e\; \mathsf{catch}\; e' \Downarrow v}$

$$\begin{array}{cl}
& \models \mathsf{try}\; e\; \mathsf{catch}\; e' : t \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& \exists v.\ \mathsf{try}\; e\; \mathsf{catch}\; e' \Downarrow v \wedge v \in [\![t]\!] \\
\Leftarrow & \{ \text{ definition of } \Downarrow \} \\
& \exists v.\ e \Downarrow v \wedge v \neq \mathsf{throw} \wedge v \in [\![t]\!] \\
\Leftrightarrow & \{ \text{ split quantifier, } \Downarrow \text{ is deterministic } \} \\
& (\exists v.\ e \Downarrow v \wedge v \in [\![t]\!]) \wedge (\exists v.\ e \Downarrow v \wedge v \neq \mathsf{throw}) \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& \models e : t \wedge (\exists v.\ e \Downarrow v \wedge v \neq \mathsf{throw})
\end{array}$$

That is, we have calculated the following semantic typing rules:

$$\frac{e \Downarrow \mathsf{throw} \qquad \models e' : t}{\models \mathsf{try}\; e\; \mathsf{catch}\; e' : t} \qquad \frac{\models e : t \qquad e \Downarrow v \qquad v \neq \mathsf{throw}}{\models \mathsf{try}\; e\; \mathsf{catch}\; e' : t}$$

We have now derived a semantic typing rule for each new evaluation rule, and can therefore read off a type system by simply replacing $\models$ by $\vdash$ in each semantic typing rule. However, the resulting type system utilises the evaluation semantics $\Downarrow$, in particular by having premises that check if an expression throws an exception or not. As before, however, we can avoid this issue by transforming the existing rules into other valid rules that do not involve evaluation.

3.2 Other Typing Rules

We start with the two derived semantic typing rules for conditionals that cover the cases when the condition throws an exception, and when it is a logical value. The transformation proceeds by first making an instantiation that brings the conclusion of each rule into the same form to allow them to be combined, and then simplifying the resulting rule:

$$\frac{e \Downarrow \mathsf{throw}}{\models \mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2 : t?} \;\wedge\; \frac{\models e : \mathsf{Bool} \qquad \models e_1 : t' \qquad \models e_2 : t'}{\models \mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2 : t'}$$

$$\Rightarrow \quad \{ \text{ instantiate } t' = t? \text{ } \}$$

$$\frac{e \Downarrow \mathsf{throw}}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t?} \quad \wedge \quad \frac{\models e : \mathsf{Bool} \qquad \models e_1 : t? \qquad \models e_2 : t?}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t?}$$

$$\Leftrightarrow \quad \{ \text{ combine into one rule } \}$$

$$\frac{e \Downarrow \mathsf{throw} \ \vee \ (\models e : \mathsf{Bool} \wedge \models e_1 : t? \wedge \models e_2 : t?)}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t?}$$

$$\Rightarrow \quad \{ \text{ bring } e \text{ terms together } \}$$

$$\frac{(e \Downarrow \mathsf{throw} \vee \models e : \mathsf{Bool}) \qquad \models e_1 : t? \qquad \models e_2 : t?}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t?}$$

$$\Leftrightarrow \quad \left\{ \text{ definition of } \models \text{ and } [\![-]\!] \right\}$$

$$\frac{\models e : \mathsf{Bool}? \qquad \models e_1 : t? \qquad \models e_2 : t?}{\models \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t?}$$

Next, we consider the semantic typing rules for addition, for which the transformation proceeds by once again making the conclusions into the same form, then combining and simplifying:

$$\frac{e \Downarrow \mathsf{throw}}{\models \mathsf{add}\ e\ e' : t?} \quad \wedge \quad \frac{\models e : \mathsf{Int} \qquad e' \Downarrow \mathsf{throw}}{\models \mathsf{add}\ e\ e' : t'?} \quad \wedge \quad \frac{\models e : \mathsf{Int} \qquad \models e' : \mathsf{Int}}{\models \mathsf{add}\ e\ e' : \mathsf{Int}}$$

$$\Rightarrow \quad \{ \text{ instantiate } t = t' = \mathsf{Int}, \text{ rule (5) with subtyping } \mathsf{Int} \le \mathsf{Int}? \text{ } \}$$

$$\frac{e \Downarrow \mathsf{throw}}{\models \mathsf{add}\ e\ e' : \mathsf{Int}?} \quad \wedge \quad \frac{\models e : \mathsf{Int} \qquad e' \Downarrow \mathsf{throw}}{\models \mathsf{add}\ e\ e' : \mathsf{Int}?} \quad \wedge \quad \frac{\models e : \mathsf{Int} \qquad \models e' : \mathsf{Int}}{\models \mathsf{add}\ e\ e' : \mathsf{Int}?}$$

$$\Leftrightarrow \quad \{ \text{ combine into one rule } \}$$

$$\frac{e \Downarrow \mathsf{throw} \ \vee \ (\models e : \mathsf{Int} \wedge e' \Downarrow \mathsf{throw}) \ \vee \ (\models e : \mathsf{Int} \wedge \models e' : \mathsf{Int})}{\models \mathsf{add}\ e\ e' : \mathsf{Int}?}$$

$$\Leftrightarrow \quad \{ \text{ factor out common term } \}$$

$$\frac{e \Downarrow \mathsf{throw} \ \vee \ (\models e : \mathsf{Int} \wedge (e' \Downarrow \mathsf{throw} \vee \models e' : \mathsf{Int}))}{\models \mathsf{add}\ e\ e' : \mathsf{Int}?}$$

$$\Leftrightarrow \quad \left\{ \text{ definition of } \models \text{ and } [\![-]\!] \right\}$$

$$\frac{e \Downarrow \mathsf{throw} \ \vee \ (\models e : \mathsf{Int} \wedge \models e' : \mathsf{Int}?)}{\models \mathsf{add}\ e\ e' : \mathsf{Int}?}$$

$$\Rightarrow \quad \{ \text{ bring } e \text{ terms together } \}$$

$$\frac{(e \Downarrow \mathsf{throw} \vee \models e : \mathsf{Int}) \qquad \models e' : \mathsf{Int}?}{\models \mathsf{add}\ e\ e' : \mathsf{Int}?}$$

$$\Leftrightarrow \quad \left\{ \text{ definition of } \models \text{ and } [\![-]\!] \right\}$$

$$\frac{\models e : \mathsf{Int}? \qquad \models e' : \mathsf{Int}?}{\models \mathsf{add}\ e\ e' : \mathsf{Int}?}$$

Finally, we consider the semantic typing rules for try/catch, from which we can derive a rule that covers the case when the first argument may throw an exception:

$$
\begin{array}{ll}
& \dfrac{\models e : t \qquad e \Downarrow v \qquad v \neq \mathsf{throw}}{\models \mathsf{try}\, e\, \mathsf{catch}\, e' : t} \;\wedge\; \dfrac{e \Downarrow \mathsf{throw} \qquad \models e' : t'}{\models \mathsf{try}\, e\, \mathsf{catch}\, e' : t'} \\
\Rightarrow & \{ \text{ instantiate } t' = t \text{ } \} \\
& \dfrac{\models e : t \qquad e \Downarrow v \qquad v \neq \mathsf{throw}}{\models \mathsf{try}\, e\, \mathsf{catch}\, e' : t} \;\wedge\; \dfrac{e \Downarrow \mathsf{throw} \qquad \models e' : t}{\models \mathsf{try}\, e\, \mathsf{catch}\, e' : t} \\
\Leftrightarrow & \{ \text{ combine into one rule } \} \\
& \dfrac{(\models e : t \wedge e \Downarrow v \wedge v \neq \mathsf{throw}) \;\vee\; (e \Downarrow \mathsf{throw} \wedge \models e' : t)}{\models \mathsf{try}\, e\, \mathsf{catch}\, e' : t} \\
\Rightarrow & \{ \text{ bring } e \text{ terms together } \} \\
& \dfrac{(\models e : t \wedge e \Downarrow v \wedge v \neq \mathsf{throw}) \vee e \Downarrow \mathsf{throw} \qquad \models e' : t}{\models \mathsf{try}\, e\, \mathsf{catch}\, e' : t} \\
\Leftrightarrow & \{ \text{ simplify first premise } \} \\
& \dfrac{\models e : t? \qquad \models e' : t}{\models \mathsf{try}\, e\, \mathsf{catch}\, e' : t}
\end{array}
$$

The final step in this calculation is justified by the following reasoning, which allows the first premise in the rule to be written in an equivalent but simpler form:

$$
\begin{array}{ll}
& \models e : t? \\
\Leftrightarrow & \{ \text{ definition of } \models \text{ } \} \\
& \exists v.\, e \Downarrow v \wedge v \in [\![t?]\!] \\
\Leftrightarrow & \left\{ \text{ definition of } [\![-]\!] \text{ } \right\} \\
& \exists v.\, e \Downarrow v \wedge ((v \in [\![t]\!] \wedge v \neq \mathsf{throw}) \vee v = \mathsf{throw}) \\
\Leftrightarrow & \{ \text{ distributivity } \} \\
& (\exists v.\, e \Downarrow v \wedge v \in [\![t]\!] \wedge v \neq \mathsf{throw}) \vee (\exists v.\, e \Downarrow v \wedge v = \mathsf{throw}) \\
\Leftrightarrow & \{ \text{ eliminate second quantifier } \} \\
& (\exists v.\, e \Downarrow v \wedge v \in [\![t]\!] \wedge v \neq \mathsf{throw}) \vee e \Downarrow \mathsf{throw} \\
\Leftrightarrow & \{ \text{ split quantifier, } \Downarrow \text{ is deterministic } \} \\
& ((\exists v.\, e \Downarrow v \wedge v \in [\![t]\!]) \wedge (\exists v.\, e \Downarrow v \wedge v \neq \mathsf{throw})) \vee e \Downarrow \mathsf{throw} \\
\Leftrightarrow & \{ \text{ definition of } \models \text{ } \} \\
& (\models e : t \wedge (\exists v.\, e \Downarrow v \wedge v \neq \mathsf{throw})) \vee e \Downarrow \mathsf{throw}
\end{array}
$$

We also get a semantic typing rule for when the first argument cannot throw an exception, which only requires transforming the second derived rule into a form that does not use evaluation:

$$
\begin{array}{ll}
& \dfrac{\models e : t \qquad e \Downarrow v \qquad v \neq \mathsf{throw}}{\models \mathsf{try}\, e\, \mathsf{catch}\, e' : t} \\
\Rightarrow & \{ \text{ transform premises } \}
\end{array}
$$

$$\frac{n \in \mathbb{Z}}{\vdash n : \mathsf{Int}} \qquad \frac{b \in \{\mathsf{false}, \mathsf{true}\}}{\vdash b : \mathsf{Bool}} \qquad \frac{}{\vdash \mathsf{throw} : t?} \qquad \frac{\vdash e : \mathsf{Int} \qquad \vdash e' : \mathsf{Int}}{\vdash \mathsf{add}\, e\, e' : \mathsf{Int}} \qquad \frac{\vdash e : \mathsf{Int}? \qquad \vdash e' : \mathsf{Int}?}{\vdash \mathsf{add}\, e\, e' : \mathsf{Int}?}$$

$$\frac{\vdash e : \mathsf{Bool} \qquad \vdash e_1 : t \qquad \vdash e_2 : t}{\vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t} \qquad \frac{\vdash e : \mathsf{Bool}? \qquad \vdash e_1 : t? \qquad \vdash e_2 : t?}{\vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t?}$$

$$\frac{\vdash e : t? \qquad \vdash e' : t}{\vdash \mathsf{try}\, e\, \mathsf{catch}\, e' : t} \qquad \frac{\vdash e : t \qquad t \text{ not of the form } t'?}{\vdash \mathsf{try}\, e\, \mathsf{catch}\, e' : t} \qquad \frac{\vdash e : t \qquad t \leq t'}{\vdash e : t'}$$

Fig. 2. Derived type system for checked exceptions.

$$\frac{\models e : t \qquad \neg(\exists t'.\, t = t'?)}{\models \mathsf{try}\, e\, \mathsf{catch}\, e' : t}$$

The transformation of the premises is justified as follows:

$$\begin{aligned}
&\models e : t \wedge e \Downarrow v \wedge v \neq \mathsf{throw} \\
\Leftrightarrow\quad &\{ \text{ definition of } \models \} \\
&(\exists v'.\, e \Downarrow v' \wedge v' \in [\![t]\!]) \wedge e \Downarrow v \wedge v \neq \mathsf{throw} \\
\Leftrightarrow\quad &\{ \Downarrow \text{ is deterministic } \} \\
&\exists v'.\, e \Downarrow v' \wedge v' \in [\![t]\!] \wedge v' \neq \mathsf{throw} \\
\Leftarrow\quad &\left\{ \text{ definition of } [\![-]\!] \right\} \\
&\exists v'.\, e \Downarrow v' \wedge v' \in [\![t]\!] \wedge \neg\exists t'.\, t = t'? \\
\Leftrightarrow\quad &\{ \text{ move first quantifier inside } \} \\
&(\exists v'.\, e \Downarrow v' \wedge v' \in [\![t]\!]) \wedge \neg\exists t'.\, t = t'? \\
\Leftrightarrow\quad &\{ \text{ definition of } \models \} \\
&\models e : t \wedge \neg(\exists t'.\, t = t'?)
\end{aligned}$$

Given the calculated semantic typing rules for $\models$, we can now simply replace $\models$ by $\vdash$ in each rule and thereby obtain an inductive definition for a type system that is, by the Knaster-Tarski theorem, guaranteed to be sound by construction. The resulting type system is given in Figure 2.

For example, according to the derived typing rules, the expression $e = \mathsf{if\ true\ then\ 1\ else\ throw}$ has type Int?, which means it either returns an integer or throws an exception, whereas $\mathsf{try}\ e\ \mathsf{catch}\ 2$ has type Int, which captures the fact that try/catch allows us to recover from the possibility of an exception being thrown in e and thereby guarantee to return an integer.

3.3 Reflection

Checked exceptions. While the simple conditional language served to introduce our approach, the extension to exceptions shows how the approach can be used to derive a non-trivial type system. In particular, we have derived a subtyping system for checked exceptions in a principled manner, starting from a type soundness property. It is also possible to derive an alternative type system that captures that an expression will certainly, rather than possibly, throw an exception. However, we chose here to derive a more subtle type system that implements checked exceptions.

Well-typed handlers. The second derived rule for try/catch allows the unused handler e' to be ill-typed if the main expression e cannot throw an exception, i.e. its type does not use ?:

$$\frac{\vdash e : t \qquad t \text{ not of the form } t'?}{\vdash \mathsf{try}\, e\, \mathsf{catch}\, e' : t}$$

This rule is perfectly valid, as all we are aiming for is type soundness, and the above guarantees this. However, if we wish to ensure handlers are always well-typed, we can derive a stronger rule that achieves this by simply adding the premise $\vdash e' : t''$, which ensures e' is well-typed.

Additional rules. One might ask why we don't need additional rules for other argument combinations, such as attempting to add expressions of type Int? and Int. The answer is that the existing rules are sufficient to derive such additional rules by subtyping. In particular, using the fact that $t \leq t?$, the following rules can be derived by subtyping from the existing rules:

$$\frac{\vdash e : \mathsf{Int?} \qquad \vdash e' : \mathsf{Int}}{\vdash \mathsf{add}\, e\, e' : \mathsf{Int?}} \qquad\qquad \frac{\vdash e : \mathsf{Int} \qquad \vdash e' : \mathsf{Int?}}{\vdash \mathsf{add}\, e\, e' : \mathsf{Int?}}$$

$$\frac{\vdash e : \mathsf{Bool?} \qquad \vdash e_1 : t \qquad \vdash e_2 : t}{\vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t?} \qquad\qquad \frac{\vdash e : t? \qquad \vdash e' : t?}{\vdash \mathsf{try}\, e\, \mathsf{catch}\, e' : t?}$$

4 Lambda Calculus

To put our methodology to the test with a more sophisticated language, we now consider a call-by-value lambda calculus extended with integers and addition:

$$e ::= n \mid \mathsf{add}\, e\, e \mid x \mid \lambda x.e \mid e\, e$$

We assume n ranges over the integers as previously, and x ranges over an (infinite) set of variable names. The previous languages were simple enough that we could have type-checked expressions by simply evaluating them and checking if the result is in a semantic type. This is not possible once we have functions, or even just free variables, both of which we have in the lambda calculus. But as we will see, the lessons learned from the previous languages still apply and allow us to calculate a type system, by refining the methodology to take account of variables.

To evaluate an expression that may contain free variables, we use a variable environment γ, a finite mapping from variables to values. We write $\mathsf{dom}(\gamma)$ for the domain of an environment γ, and $\gamma[x \mapsto v]$ for the environment γ extended with the mapping $x \mapsto v$. A value is either an integer n or a function closure consisting of an environment γ and a lambda abstraction $\lambda x.e$:

$$v ::= n \mid \langle \gamma, \lambda x.e \rangle$$

The semantics of the language is given by an evaluation relation $e \Downarrow_\gamma v$ that specifies when an expression e evaluates in an environment γ to a value v, defined by the following rules:

$$\frac{}{n \Downarrow_\gamma n} \qquad \frac{e \Downarrow_\gamma n \qquad e' \Downarrow_\gamma n'}{\mathsf{add}\, e\, e' \Downarrow_\gamma n + n'} \qquad \frac{x \in \mathsf{dom}(\gamma)}{x \Downarrow_\gamma \gamma(x)}$$

$$\frac{}{\lambda x.e \Downarrow_\gamma \langle \gamma, \lambda x.e \rangle} \qquad \frac{e \Downarrow_\gamma \langle \gamma', \lambda x.e'' \rangle \qquad e' \Downarrow_\gamma v \qquad e'' \Downarrow_{\gamma'[x \mapsto v]} w}{e\, e' \Downarrow_\gamma w}$$

The rules for integers and addition are similar to previously, with the addition of an environment. Variables are simply looked up in the environment, with a premise to ensure they are defined. In turn, abstractions are evaluated to a closure $\langle \gamma, \lambda x.e \rangle$, which captures the environment γ in which the abstraction is evaluated, the variable x that is bound by the abstraction, and the body e of the abstraction. And finally, when applying a function, we first evaluate it to such a closure with an environment γ', and then evaluate the body of the closure in an extended environment $\gamma'[x \mapsto v]$, where v is the value that was passed as an argument to the function.

The language of types consists of integers and function types:

$$t ::= \mathsf{Int} \mid t \rightarrow t$$

The semantic type corresponding to Int is, as before, the set of integers $\mathbb{Z}$. The semantic type of functions from s to t formalises the intuition that a closure of this type must be able to take any value of semantic type $[\![s]\!]$ and produce a value of semantic type $[\![t]\!]$:

$$[\![\mathsf{Int}]\!] = \mathbb{Z} \qquad\qquad [\![s \rightarrow t]\!] = \left\{ \langle \gamma, \lambda x.e \rangle \,\middle|\, \forall v \in [\![s]\!] \,.\, \exists w.\ e \Downarrow_{\gamma[x \mapsto v]} w \ \wedge\ w \in [\![t]\!] \right\}$$

The definition for the semantic function type $[\![s \rightarrow t]\!]$ is standard, but we will return to this in the reflection section and show how it can also be calculated if desired.

Because the evaluation relation now takes an environment γ, the semantic typing relation $\models$ must also account for γ, which we achieve in two steps. We first define an *evaluation typing relation* $\gamma \Mapsto e : t$, which states that e evaluates to a value of semantic type t in an environment γ:

$$\gamma \Mapsto e : t \qquad \stackrel{\text{def}}{\Longleftrightarrow} \qquad \exists v.\ e \Downarrow_\gamma v \wedge v \in [\![t]\!]$$

This relation only partially abstracts from concrete values to semantic types. In particular, the value v produced by an evaluation $e \Downarrow_\gamma v$ is abstracted to the semantic type $[\![t]\!]$, but the environment γ still remains. To abstract from γ as well, we extend the definition of semantic types to semantic typing contexts. A (syntactic) typing context Γ is a finite mapping from variables to types, typically written in the form $x_1 : t_1, \ldots, x_n : t_n$. For each typing context Γ, we define a corresponding semantic typing context $[\![\Gamma]\!]$ by recursion on the structure of Γ:

$$[\![\cdot]\!] = \{\emptyset\} \qquad\qquad [\![\Gamma, x : t]\!] = \left\{ \gamma[x \mapsto v] \,\middle|\, \gamma \in [\![\Gamma]\!] \wedge v \in [\![t]\!] \right\}$$

The empty typing context $\cdot$ is assigned the semantic typing context $\{\emptyset\}$, which only contains the empty environment $\emptyset$. The extension of a typing context Γ with a variable $x : t$, written $\Gamma, x : t$, is represented in the semantic typing context with corresponding environments γ extended with mappings of the form $x \mapsto v$. We can characterise semantic typing contexts more succinctly as consisting of those environments that map variables to semantically well-typed values:

$$x : t \in \Gamma \quad \Rightarrow \quad \gamma(x) \in [\![t]\!] \qquad \text{for all } \gamma \in [\![\Gamma]\!] \tag{6}$$

This property follows by a straightforward induction on Γ. Finally, we can now define the semantic typing relation in terms of these semantic typing contexts,

$$\Gamma \models e : t \qquad \stackrel{\text{def}}{\Longleftrightarrow} \qquad \forall \gamma \in [\![\Gamma]\!] \,.\ \gamma \Mapsto e : t$$

and the desired type soundness property can then be formulated in the same way as previously, with the addition of a typing context to deal with variables:

$$\frac{\Gamma \vdash e : t}{\Gamma \models e : t}$$

4.1 Calculating a Type System

We could now calculate semantic typing rules in a manner similar to previously. However, we use a slightly different approach here, which simplifies some calculations and which will prove essential for combining rules. Instead of directly using the semantic typing relation $\Gamma \models e : t$, we first use the evaluation typing relation $\gamma \Rrightarrow e : t$ and aim to derive properties of the form

$$\frac{Q}{\gamma \Rrightarrow e : t} \tag{7}$$

where the premise Q refers to $\Rrightarrow$ only in strictly positive positions, i.e. neither subject to negation nor occurring on the left-hand side of an implication. We refer to a property of the above form as an *evaluation typing rule* or simply as a *rule* for $\Rrightarrow$. Only afterwards do we perform a final abstraction step from variable environments to typing contexts to obtain semantic typing rules of the following desired form, where P is any additional premise that does not refer to $\models$.

$$\frac{\Gamma \models e_1 : t_1 \quad \dots \quad \Gamma \models e_n : t_n \qquad P}{\Gamma \models e : t} \tag{8}$$

The form (7) for derived rules for $\Rrightarrow$ is intentionally more permissive than the form (8) for $\models$. The derived rules for $\Rrightarrow$ are an *intermediate* result from which the rules for $\models$ are subsequently derived via further calculation. The stricter form for rules for $\models$ ensures that they conform with the requirements of our methodology, which we use to obtain the sound-by-construction typing rules. While this two-step process is not strictly necessary for the lambda calculus language we consider here, it will be crucial for more complex languages as we will see in section 5.

The calculation proceeds again by considering each rule of the evaluation semantics, seeking to strengthen a term of the form $\gamma \Rrightarrow e : t$, where e is the expression being evaluated, into a form that matches the premise of (7). Because the semantics for integers and addition are essentially the same as previously, the calculations for these rules are also essentially the same, apart from the addition of an environment, and result in the following evaluation typing rules:

$$\frac{}{\gamma \Rrightarrow n : \mathsf{Int}} \qquad\qquad \frac{\gamma \Rrightarrow e : \mathsf{Int} \qquad \gamma \Rrightarrow e' : \mathsf{Int}}{\gamma \Rrightarrow \mathsf{add}\, e\, e' : \mathsf{Int}}$$

The new cases for variables and lambda abstraction are straightforward.

Case: $\dfrac{x \in \mathsf{dom}(\gamma)}{x \Downarrow_\gamma \gamma(x)}$

$$\begin{aligned}
& \gamma \Rrightarrow x : t \\
\Leftrightarrow \quad & \{ \text{ definition of } \Rrightarrow \} \\
& \exists v.\ x \Downarrow_\gamma v \wedge v \in [\![t]\!] \\
\Leftrightarrow \quad & \{ \text{ definition of } \Downarrow \} \\
& \gamma(x) \in [\![t]\!]
\end{aligned}$$

Case: $\dfrac{}{\lambda x.e \Downarrow_\gamma \langle \gamma, \lambda x.e \rangle}$

$$\begin{aligned}
& \gamma \Rrightarrow \lambda x.e : t \\
\Leftrightarrow \quad & \{ \text{ definition of } \Rrightarrow \} \\
& \exists v.\ \lambda x.e \Downarrow_\gamma v \wedge v \in [\![t]\!] \\
\Leftrightarrow \quad & \{ \text{ definition of } \Downarrow \} \\
& \langle \gamma, \lambda x.e \rangle \in [\![t]\!] \\
\Leftrightarrow \quad & \{ \text{ definition of } [\![-]\!] \} \\
& \exists t_1, t_2.\ t = t_1 \to t_2 \wedge \forall v \in [\![t_1]\!] .\ \exists w.\ e \Downarrow_{\gamma[x \mapsto v]} w \wedge w \in [\![t_2]\!] \\
\Leftrightarrow \quad & \{ \text{ definition of } \Rrightarrow \} \\
& \exists t_1, t_2.\ t = t_1 \to t_2 \wedge \forall v \in [\![t_1]\!] .\ \gamma[x \mapsto v] \Rrightarrow e : t_2
\end{aligned}$$

After the usual simplification that incorporates the equality premise $t = t_1 \to t_2$ into the conclusion, we have calculated the following rules, both of which have the desired form (7):

$$\frac{\gamma(x) \in [\![t]\!]}{\gamma \Rrightarrow x : t} \qquad\qquad \frac{\forall v \in [\![t_1]\!] .\ \gamma[x \mapsto v] \Rrightarrow e : t_2}{\gamma \Rrightarrow \lambda x.e : t_1 \to t_2}$$

All that remains now is the case for function application.

Case: $$\frac{e \Downarrow_\gamma \langle \gamma', \lambda x.e'' \rangle \qquad e' \Downarrow_\gamma v \qquad e'' \Downarrow_{\gamma'[x \mapsto v]} w}{e\, e' \Downarrow_\gamma w}$$

$$\begin{array}{ll}
& \gamma \Rrightarrow e\, e' : t \\
\Leftrightarrow & \{ \text{ definition of } \Rrightarrow \} \\
& \exists w.\ e\, e' \Downarrow_\gamma w \wedge w \in [\![t]\!] \\
\Leftrightarrow & \{ \text{ definition of } \Downarrow \} \\
& \exists w, \gamma', x, e'', v.\ e \Downarrow_\gamma \langle \gamma', \lambda x.e'' \rangle \wedge e' \Downarrow_\gamma v \wedge e'' \Downarrow_{\gamma'[x \mapsto v]} w \wedge w \in [\![t]\!]
\end{array}$$

After applying the definition of $\Rrightarrow$ and $\Downarrow$ as in the other cases, we appear to be stuck due to the last two conjuncts $e'' \Downarrow_{\gamma'[x \mapsto v]} w$ and $w \in [\![t]\!]$. Because both depend on e'' and γ', they are not suitable to be included in an evaluation typing rule. In order to discharge these conjuncts, we extend the statement with additional assumptions about the semantic type of the closure $\langle \gamma', \lambda x.e'' \rangle$ and the value v. Taken together, these two assumptions will precisely discharge the two conjuncts noted above: given $\langle \gamma', \lambda x.e'' \rangle \in [\![s \to t]\!]$ and $v \in [\![s]\!]$, the definition of $[\![s \to t]\!]$ implies that $e'' \Downarrow_{\gamma'[x \mapsto v]} w$ and $w \in [\![t]\!]$. With this insight, we can resume and complete the calculation:

$$\begin{array}{ll}
& \exists w, \gamma', x, e'', v.\ e \Downarrow_\gamma \langle \gamma', \lambda x.e'' \rangle \wedge e' \Downarrow_\gamma v \wedge e'' \Downarrow_{\gamma'[x \mapsto v]} w \wedge w \in [\![t]\!] \\
\Leftarrow & \left\{ \text{ strengthen with } \langle \gamma', \lambda x.e'' \rangle \in [\![s \to t]\!] \text{ and } v \in [\![s]\!] \right\} \\
& \exists w, \gamma', x, e'', v, s.\ e \Downarrow_\gamma \langle \gamma', \lambda x.e'' \rangle \wedge \langle \gamma', \lambda x.e'' \rangle \in [\![s \to t]\!] \wedge e' \Downarrow_\gamma v \wedge v \in [\![s]\!] \\
& \qquad \wedge\ e'' \Downarrow_{\gamma'[x \mapsto v]} w \wedge w \in [\![t]\!] \\
\Leftrightarrow & \{ \text{ move quantifier for } w \text{ inside } \} \\
& \exists \gamma', x, e'', v, s.\ e \Downarrow_\gamma \langle \gamma', \lambda x.e'' \rangle \wedge \langle \gamma', \lambda x.e'' \rangle \in [\![s \to t]\!] \wedge e' \Downarrow_\gamma v \wedge v \in [\![s]\!] \\
& \qquad \wedge\ \exists w.\ e'' \Downarrow_{\gamma'[x \mapsto v]} w \wedge w \in [\![t]\!] \\
\Leftrightarrow & \left\{ \text{ by definition, } \langle \gamma', \lambda x.e'' \rangle \in [\![s \to t]\!] \text{ and } v \in [\![s]\!] \text{ imply } \exists w.\ e'' \Downarrow_{\gamma'[x \mapsto v]} w \wedge w \in [\![t]\!] \right\} \\
& \exists \gamma', x, e'', v, s.\ e \Downarrow_\gamma \langle \gamma', \lambda x.e'' \rangle \wedge \langle \gamma', \lambda x.e'' \rangle \in [\![s \to t]\!] \wedge e' \Downarrow_\gamma v \wedge v \in [\![s]\!] \\
\Leftrightarrow & \{ \text{ move quantifiers inside } \} \\
& \exists s.\ (\exists \gamma', x, e''.\ e \Downarrow_\gamma \langle \gamma', \lambda x.e'' \rangle \wedge \langle \gamma', \lambda x.e'' \rangle \in [\![s \to t]\!]) \wedge \exists v.\ e' \Downarrow_\gamma v \wedge v \in [\![s]\!] \\
\Leftrightarrow & \left\{ \text{ definition of } \Rrightarrow \text{ and } [\![s \to t]\!] \right\} \\
& \exists s.\ \gamma \Rrightarrow e : s \to t \wedge \gamma \Rrightarrow e' : s
\end{array}$$

After an additional calculation step that moves the existential quantifier out of the premise, we have calculated the following evaluation typing rule:

$$\frac{\gamma \Rrightarrow e : s \to t \qquad \gamma \Rrightarrow e' : s}{\gamma \Rrightarrow e\, e' : t}$$

Finally, we take the derived evaluation typing rules and abstract from the environment γ to obtain semantic typing rules. Three of the calculated evaluation typing rules, namely for integers, addition, and function application, are of the form:

$$\frac{\gamma \Rrightarrow e_1 : t_1 \quad \ldots \quad \gamma \Rrightarrow e_n : t_n}{\gamma \Rrightarrow e : t} \tag{9}$$

Each such evaluation typing rule implies a corresponding semantic typing rule of form (3).

$$\frac{\Gamma \models e_1 : t_1 \quad \dots \quad \Gamma \models e_n : t_n}{\Gamma \models e : t}$$

To prove this transformation step sound, we assume (9) and $\Gamma \models e_1 : t_1 \wedge \dots \wedge \Gamma \models e_n : t_n$, and we must show that $\Gamma \models e : t$. To this end, we assume some environment $\gamma \in [\![\Gamma]\!]$, and must show that $\gamma \Mapsto e : t$. Because $\Gamma \models e_i : t_i$ and $\gamma \in [\![\Gamma]\!]$, we have by the definition of the semantic typing relation $\models$ that $\gamma \Mapsto e_i : t_i$. Hence, by (9), we have that $\gamma \Mapsto e : t$.

Applying this general transformation gives us the following semantic typing rules:

$$\frac{}{\Gamma \models n : \mathsf{Int}} \qquad \frac{\Gamma \models e : \mathsf{Int} \qquad \Gamma \models e' : \mathsf{Int}}{\Gamma \models \mathsf{add}\, e\, e' : \mathsf{Int}} \qquad \frac{\Gamma \models e : s \to t \qquad \Gamma \models e' : s}{\Gamma \models e\, e' : t}$$

The remaining two calculated evaluation typing rules, namely the rules for variables and lambda abstraction, do not match the form (9), as we can see here:

$$\frac{\gamma(x) \in [\![t]\!]}{\gamma \Mapsto x : t} \quad (10) \qquad\qquad \frac{\forall v \in [\![t_1]\!] \,.\, \gamma[x \mapsto v] \Mapsto e : t_2}{\gamma \Mapsto \lambda x.e : t_1 \to t_2} \quad (11)$$

However, in both cases, we can easily calculate corresponding semantic typing rules. We start with the conclusion of the desired semantic typing rule and transform it step by step until we reach a form suitable for the premise of a semantic typing rule of the form (8):

$$\begin{array}{ll}
& \Gamma \models x : t \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& \forall \gamma \in [\![\Gamma]\!] \,.\, \gamma \Mapsto x : t \\
\Leftarrow & \{ \text{ evaluation typing rule (10) } \} \\
& \forall \gamma \in [\![\Gamma]\!] \,.\, \gamma(x) \in [\![t]\!] \\
\Leftarrow & \{ \text{ property (6) } \} \\
& \forall \gamma \in [\![\Gamma]\!] \,.\, x : t \in \Gamma \\
\Leftrightarrow & \left\{ \text{ eliminate quantifier, } [\![\Gamma]\!] \text{ is non-empty } \right\} \\
& x : t \in \Gamma
\end{array}$$

$$\begin{array}{ll}
& \Gamma \models \lambda x.e : t_1 \to t_2 \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& \forall \gamma \in [\![\Gamma]\!] \,.\, \gamma \Mapsto \lambda x.e : t_1 \to t_2 \\
\Leftarrow & \{ \text{ evaluation typing rule (11) } \} \\
& \forall \gamma \in [\![\Gamma]\!] \,.\, \forall v \in [\![t_1]\!] \,.\gamma[x \mapsto v] \Mapsto e : t_2 \\
\Leftrightarrow & \left\{ \text{ definition of } [\![\Gamma, x : t_1]\!] \right\} \\
& \forall \gamma' \in [\![\Gamma, x : t_1]\!] \,.\, \gamma' \Mapsto e : t_2 \\
\Leftrightarrow & \{ \text{ definition of } \models \} \\
& \Gamma, x : t_1 \models e : t_2
\end{array}$$

That is, we have obtained the following semantic typing rules:

$$\frac{x : t \in \Gamma}{\Gamma \models x : t} \qquad\qquad \frac{\Gamma, x : t_1 \models e : t_2}{\Gamma \models \lambda x.e : t_1 \to t_2}$$

Given the calculated semantic typing rules, we now obtain sound-by-construction syntactic typing rules for our language by simply replacing $\models$ with $\vdash$, as summarised in Figure 3. These are the standard typing rules for simply typed lambda calculus. But while the rules are standard, the way we obtained them is not, namely by systematic calculation that guarantees soundness. The same systematic calculation technique applies to languages with more complex combinations of features where designing a type system by hand is more challenging and would still require a separate soundness proof. We will see an example of this in section 5.

4.2 Reflection

The calculation of the semantic typing rule for application hinges on the definition of the semantic function type $[\![s \to t]\!]$. While the definition we used is standard, it is interesting to consider if it

$$\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e : \mathsf{Int} \quad \Gamma \vdash e' : \mathsf{Int}}{\Gamma \vdash \mathsf{add}\, e\, e' : \mathsf{Int}} \qquad \frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad \frac{\Gamma \vdash e : s \to t \quad \Gamma \vdash e' : s}{\Gamma \vdash e\, e' : t}$$

Fig. 3. Derived type system for the lambda calculus.

could have been discovered during the calculation, rather than given upfront. If possible, this may benefit the calculation for more complex languages.

Using the idea of partial specifications from earlier work on compiler calculation [Bahr and Hutton 2015], we could have started the calculation with a partial definition

$$[\![s \to t]\!] \;=\; \{\langle \gamma, \lambda x.e\rangle \mid P(\gamma, x, e, s, t)\}$$

where the predicate P is left unspecified. We then aim for the definition of P to be discovered during the calculation. In fact, during the calculation for application, we run into the problem of having to discharge the conjuncts $e'' \Downarrow_{\gamma'[x\mapsto w]} v$ and $v \in [\![t]\!]$. The strategy to solve this problem is to strengthen the statement to include the assumption $\langle \gamma', \lambda x.e''\rangle \in [\![s \to t]\!]$ and then use P to discharge the two conjuncts. This is achieved by defining P to give exactly these conjuncts:

$$P(\gamma, x, e, s, t) \quad \overset{\text{def}}{\iff} \quad \forall w.\ \exists v.\ e \Downarrow_{\gamma[x\mapsto w]} v \wedge v \in [\![t]\!] \tag{12}$$

This definition for P allows the calculation for function application to go through, even though it ignores the domain type s of the function. However, we would get stuck in the calculation for lambda abstraction, which needs w to be semantically well-typed, i.e. $w \in [\![s']\!]$ for some type s'. An obvious choice would be to instantiate $s' = s$, i.e. to refine P so that it assumes $w \in [\![s]\!]$:

$$P(\gamma, x, e, s, t) \quad \overset{\text{def}}{\iff} \quad \forall w \in [\![s]\!]\ .\ \exists v.\ e \Downarrow_{\gamma[x\mapsto w]} v \wedge v \in [\![t]\!]$$

This definition now matches the semantic type definition used in our calculation in section 4.1 and results in the standard typing rules for simply typed lambda calculus. But we could have also chosen a different definition of P that just assumes w is semantically typed by any type s':

$$P(\gamma, x, e, s, t) \quad \overset{\text{def}}{\iff} \quad \forall s'.\ \forall w \in [\![s']\!]\ .\ \exists v.\ e \Downarrow_{\gamma[x\mapsto w]} v \wedge v \in [\![t]\!] \tag{13}$$

With this definition of P, the calculations for abstraction and application go through, but they result in a different type system that requires functions to take values of any type as argument:

$$\frac{\Gamma \vdash e : s \to t \quad \Gamma \vdash e' : s'}{\Gamma \vdash e\, e' : t} \qquad\qquad \frac{\forall s'.\ \Gamma, x : s' \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t}$$

This type system is not only unusual, but also not very useful: it effectively prohibits the use of any function argument, because no assumption can be made about its type.

The above reasoning shows that the calculational approach does allow us to discover the definition of the semantic function type. But this process may require some iteration, because we might discover a definition that leads to a dead end, such as with (12), or to an impractical type system, such as with (13). Such an iterative process is typical in devising type soundness proofs, but with an important difference: in typical type soundness proofs, adjustments often have to be made both on the typing rules and the semantic type definition, with the hope that the adjustments will lead to a soundness proof. In the calculational approach, we only need to adjust the definition of the semantic type, and we then let calculation lead us to sound-by-construction typing rules.

Syntax:

$$e ::= n \mid b \mid \mathsf{throw} \mid \mathsf{add}\, e\, e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{try}\, e\, \mathsf{catch}\, e \mid x \mid \lambda x.e \mid e\, e$$

$$v ::= n \mid b \mid \mathsf{throw} \mid \langle \gamma, \lambda x.e \rangle$$

$$t ::= \mathsf{Int} \mid \mathsf{Bool} \mid t? \mid t \rightarrow t$$

Evaluation Semantics:

$$\frac{}{n \Downarrow_\gamma n} \qquad \frac{}{b \Downarrow_\gamma b} \qquad \frac{}{\mathsf{throw} \Downarrow_\gamma \mathsf{throw}} \qquad \frac{e \Downarrow_\gamma n \quad e' \Downarrow_\gamma n'}{\mathsf{add}\, e\, e' \Downarrow_\gamma n + n'} \qquad \frac{e \Downarrow_\gamma \mathsf{throw}}{\mathsf{add}\, e\, e' \Downarrow_\gamma \mathsf{throw}} \qquad \frac{e \Downarrow_\gamma n \quad e' \Downarrow_\gamma \mathsf{throw}}{\mathsf{add}\, e\, e' \Downarrow_\gamma \mathsf{throw}}$$

$$\frac{e \Downarrow_\gamma \mathsf{true} \quad e_1 \Downarrow_\gamma v}{\mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \Downarrow_\gamma v} \qquad \frac{e \Downarrow_\gamma \mathsf{false} \quad e_2 \Downarrow_\gamma v}{\mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \Downarrow_\gamma v} \qquad \frac{e \Downarrow_\gamma \mathsf{throw}}{\mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \Downarrow_\gamma \mathsf{throw}}$$

$$\frac{e \Downarrow_\gamma \mathsf{throw} \quad e' \Downarrow_\gamma v}{\mathsf{try}\, e\, \mathsf{catch}\, e' \Downarrow_\gamma v} \qquad \frac{e \Downarrow_\gamma v \quad v \neq \mathsf{throw}}{\mathsf{try}\, e\, \mathsf{catch}\, e' \Downarrow_\gamma v}$$

$$\frac{x \in \mathrm{dom}(\gamma)}{x \Downarrow_\gamma \gamma(x)} \qquad \frac{}{\lambda x.e \Downarrow_\gamma \langle \gamma, \lambda x.e \rangle} \qquad \frac{e \Downarrow_\gamma \langle \gamma', \lambda x.e'' \rangle \quad e' \Downarrow_\gamma v \quad v \neq \mathsf{throw} \quad e'' \Downarrow_{\gamma'[x \mapsto v]} w}{e\, e' \Downarrow_\gamma w}$$

$$\frac{e \Downarrow_\gamma \mathsf{throw}}{e\, e' \Downarrow_\gamma \mathsf{throw}} \qquad \frac{e \Downarrow_\gamma \langle \gamma', \lambda x.e'' \rangle \quad e' \Downarrow_\gamma \mathsf{throw}}{e\, e' \Downarrow_\gamma \mathsf{throw}}$$

Semantic Types:

$$[\![\mathsf{Int}]\!] = \mathbb{Z} \qquad [\![\mathsf{Bool}]\!] = \{\mathsf{false}, \mathsf{true}\} \qquad [\![t?]\!] = [\![t]\!] \cup \{\mathsf{throw}\}$$

$$[\![s \rightarrow t]\!] = \left\{ \langle \gamma, \lambda x.e \rangle \,\middle|\, \forall v \in [\![s]\!] \,.\, \exists w. e \Downarrow_{\gamma[x \mapsto v]} w \wedge w \in [\![t]\!] \right\}$$

Semantic Contexts:

$$[\![\cdot]\!] = \{\emptyset\} \qquad [\![\Gamma, x : t]\!] = \left\{ \gamma[x \mapsto v] \,\middle|\, \gamma \in [\![\Gamma]\!] \wedge v \in [\![t]\!] \right\}$$

Fig. 4. Syntax and semantics of lambda calculus with conditionals and checked exceptions.

5 Lambda Calculus with Conditionals and Checked Exceptions

For our final example, we consider the lambda calculus from section 4 extended with conditionals and checked exceptions in the style of section 3. The syntax and semantics are given in Figure 4.

Designing typing rules that soundly capture the combination of higher-order functions and checked exceptions is more challenging than for languages with only one of these features. In contrast, the calculational approach is systematic and modular, and handles this combination gracefully. To calculate a sound-by-construction type system for this language, we combine the techniques we have learned so far. In fact, the modular reasoning enabled by the *semantic* approach to type soundness allows us to *reuse* previous calculations with little to no modifications.

As with the lambda calculus example in section 4, we define an evaluation typing relation and a semantic typing relation to capture the desired soundness property:

$$\gamma \Mapsto e : t \quad \stackrel{\text{def}}{\iff} \quad \exists v.\, e \Downarrow_\gamma v \wedge v \in [\![t]\!]$$

$$\Gamma \models e : t \quad \stackrel{\text{def}}{\iff} \quad \forall \gamma \in [\![\Gamma]\!] \,.\, \gamma \Mapsto e : t$$

As previously, we aim to calculate evaluation typing rules of the form

$$\frac{P}{\gamma \Mapsto e : t}$$

where P refers to $\Mapsto$ only in strictly positive positions. Once these rules have been calculated, we can combine them as in sections 2.4 and 3.2 to obtain evaluation typing rules of the desired form, and then finally transform them into semantic typing rules as in section 4.

We proceed by considering each rule of the evaluation semantics. As observed earlier, the calculations for integers and addition in section 4 are essentially the same as the calculations in section 3. Indeed, all calculations in section 3 can be performed for the extended language as well and result in essentially the same rules, except that we use the evaluation typing relation $\gamma \Mapsto e : t$ instead of the semantic typing relation $\models e : t$, and the rules refer to a variable environment γ. For instance, we can calculate the following evaluation typing rules for conditional expressions:

$$\frac{e \Downarrow_\gamma \mathsf{true} \qquad \gamma \Mapsto e_1 : t}{\gamma \Mapsto \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t} \qquad \frac{e \Downarrow_\gamma \mathsf{false} \qquad \gamma \Mapsto e_2 : t}{\gamma \Mapsto \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t} \qquad \frac{e \Downarrow_\gamma \mathsf{throw}}{\gamma \Mapsto \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t?} \tag{14}$$

Here it is important that the calculations use the evaluation typing relation $\gamma \Mapsto e : t$ rather than the semantic typing relation $\Gamma \models e : t$. In particular, all three of the above rules depend on the fact that we can directly refer to the variable environment γ in the premise, which allows us to state how the expression e evaluates in the environment γ. If we had used semantic typing, the calculations would have resulted in the following rules instead:

$$\frac{\forall \gamma \in [\![\Gamma]\!].\, e \Downarrow_\gamma \mathsf{true} \quad \Gamma \vdash e_1 : t}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t} \qquad \frac{\forall \gamma \in [\![\Gamma]\!].\, e \Downarrow_\gamma \mathsf{false} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t} \qquad \frac{\forall \gamma \in [\![\Gamma]\!].\, e \Downarrow_\gamma \mathsf{throw}}{\Gamma \models \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t?}$$

These rules are far more restrictive. For example, the last requires e to throw an exception in *every* semantically well-typed environment γ. As a consequence, these rules could not have been used to derive further rules as in section 3.2. However, with the rules in (14) we can apply the same reasoning steps as previously to calculate the following rules:

$$\frac{\gamma \Mapsto e : \mathsf{Bool?} \qquad \gamma \Mapsto e_1 : t? \qquad \gamma \Mapsto e_2 : t?}{\gamma \Mapsto \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t?}$$

All calculations from section 4 also carry over to our extended lambda calculus, except that the $\Leftrightarrow$-steps that apply the definition of the evaluation relation become $\Leftarrow$-steps here, because more than one evaluation rule may now apply to expressions of the same shape. We therefore obtain the same evaluation typing rules for variables, abstraction and application as previously:

$$\frac{\gamma(x) \in [\![t]\!]}{\gamma \Mapsto x : t} \qquad \frac{\forall v \in [\![t_1]\!] \,.\, \gamma[x \mapsto v] \Mapsto e : t_2}{\gamma \Mapsto \lambda x.e : t_1 \to t_2} \qquad \frac{\gamma \Mapsto e : s \to t \qquad \gamma \Mapsto e' : s}{\gamma \Mapsto e\, e' : t}$$

The only rules of the semantics for which we cannot directly reuse previous calculations are the three rules for application. Two of these rules deal with the propagation of exceptions and thus do not have direct counterparts in our earlier languages. But the calculations for these two cases are straightforward, by simply applying definitions, and result in the following rules:

$$\frac{e \Downarrow_\gamma \mathsf{throw}}{\gamma \Mapsto e\, e' : t?} \qquad \frac{\gamma \Mapsto e : t_1 \to t_2 \qquad e' \Downarrow_\gamma \mathsf{throw}}{\gamma \Mapsto e\, e' : t?} \tag{15}$$

The remaining rule for the semantics of application is similar to the corresponding rule in section 4 but has the additional side condition $v \neq \mathsf{throw}$. We can therefore reuse the calculation from that section with only minor changes that simply carry over the side condition, similarly to the calculation for try/catch in section 3, which results in the following rule:

$$\frac{\gamma \Mapsto e : s \to t \qquad \gamma \Mapsto e' : s \qquad e' \Downarrow_\gamma v \qquad v \neq \mathsf{throw}}{\gamma \Mapsto e\, e' : t} \tag{16}$$

$$
\begin{array}{l}
\dfrac{e \Downarrow_\gamma \mathsf{throw}}{\gamma \Rrightarrow e\, e' : t?} \wedge \dfrac{\gamma \Rrightarrow e : t_1 \to t_2 \quad e' \Downarrow_\gamma \mathsf{throw}}{\gamma \Rrightarrow e\, e' : t_3?} \wedge \dfrac{\gamma \Rrightarrow e : s_1 \to s_2 \quad \gamma \Rrightarrow e' : s_1 \quad e' \Downarrow_\gamma v \quad v \neq \mathsf{throw}}{\gamma \Rrightarrow e\, e' : s_2} \\
\Rightarrow \quad \{ \text{ instantiate } t_3 = t \text{ and } s_2 = t? \; \} \\
\dfrac{e \Downarrow_\gamma \mathsf{throw}}{\gamma \Rrightarrow e\, e' : t?} \wedge \dfrac{\gamma \Rrightarrow e : t_1 \to t_2 \quad e' \Downarrow_\gamma \mathsf{throw}}{\gamma \Rrightarrow e\, e' : t?} \wedge \dfrac{\gamma \Rrightarrow e : s_1 \to t? \quad \gamma \Rrightarrow e' : s_1 \quad e' \Downarrow_\gamma v \quad v \neq \mathsf{throw}}{\gamma \Rrightarrow e\, e' : t?} \\
\Leftrightarrow \quad \{ \text{ combine into one rule } \} \\
\dfrac{e \Downarrow_\gamma \mathsf{throw} \vee (\gamma \Rrightarrow e : t_1 \to t_2 \wedge e' \Downarrow_\gamma \mathsf{throw}) \vee (\gamma \Rrightarrow e : s_1 \to t? \wedge \gamma \Rrightarrow e' : s_1 \wedge e' \Downarrow_\gamma v \wedge v \neq \mathsf{throw})}{\gamma \Rrightarrow e\, e' : t?} \\
\Rightarrow \quad \{ \text{ instantiate } t_1 = s_1 \text{ and } t_2 = t? \; \} \\
\dfrac{e \Downarrow_\gamma \mathsf{throw} \vee (\gamma \Rrightarrow e : s_1 \to t? \wedge e' \Downarrow_\gamma \mathsf{throw}) \vee (\gamma \Rrightarrow e : s_1 \to t? \wedge \gamma \Rrightarrow e' : s_1 \wedge e' \Downarrow_\gamma v \wedge v \neq \mathsf{throw})}{\gamma \Rrightarrow e\, e' : t?} \\
\Leftrightarrow \quad \{ \text{ distributivity of } \wedge \text{ over } \vee \; \} \\
\dfrac{e \Downarrow_\gamma \mathsf{throw} \vee (\gamma \Rrightarrow e : s_1 \to t? \wedge (e' \Downarrow_\gamma \mathsf{throw} \vee (\gamma \Rrightarrow e' : s_1 \wedge e' \Downarrow_\gamma v \wedge v \neq \mathsf{throw})))}{\gamma \Rrightarrow e\, e' : t?} \\
\Leftrightarrow \quad \left\{ \text{ definition of } [\![-]\!] \text{ and } \Rrightarrow \right\} \\
\dfrac{e \Downarrow_\gamma \mathsf{throw} \vee (\gamma \Rrightarrow e : s_1 \to t? \wedge \gamma \Rrightarrow e' : s_1?)}{\gamma \Rrightarrow e\, e' : t?} \\
\Rightarrow \quad \{ \text{ bring } e \text{ terms together } \} \\
\dfrac{(e \Downarrow_\gamma \mathsf{throw} \vee \gamma \Rrightarrow e : s_1 \to t?) \wedge \gamma \Rrightarrow e' : s_1?}{\gamma \Rrightarrow e\, e' : t?} \\
\Leftrightarrow \quad \left\{ \text{ definition of } [\![-]\!] \text{ and } \Rrightarrow \right\} \\
\dfrac{\gamma \Rrightarrow e : (s_1 \to t?)? \wedge \gamma \Rrightarrow e' : s_1?}{\gamma \Rrightarrow e\, e' : t?}
\end{array}
$$

Fig. 5. Calculation to combine evaluation typing rules for application.

The resulting side condition that e' evaluate to a value different from throw reminds us of a similar condition found in the semantic typing rules for try/catch in section 3. Using the same calculation argument as in section 3.2, we can derive an evaluation typing rule that replaces this side condition on the evaluation result with a side condition on the type of e':

$$\frac{\gamma \Rrightarrow e : s \to t \qquad \gamma \Rrightarrow e' : s \qquad s \text{ not of the form } s'?}{\gamma \Rrightarrow e\, e' : t}$$

This rule is suitable to be turned into a semantic typing rule. However, the original rule (16) can be used in conjunction with the other two rules for function application in (15) to calculate an evaluation typing rule for application in the presence of exceptions as shown in Figure 5. That is, we can derive the following evaluation typing rule:

$$\frac{\gamma \Rrightarrow e : (s \to t?)? \qquad \gamma \Rrightarrow e' : s?}{\gamma \Rrightarrow e\, e' : t?}$$

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash b : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{throw} : t?} \qquad \frac{\Gamma \vdash e : \mathsf{Int} \quad \Gamma \vdash e' : \mathsf{Int}}{\Gamma \vdash \mathsf{add}\, e\, e' : \mathsf{Int}} \qquad \frac{\Gamma \vdash e : \mathsf{Int?} \quad \Gamma \vdash e' : \mathsf{Int?}}{\Gamma \vdash \mathsf{add}\, e\, e' : \mathsf{Int?}}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t} \qquad \frac{\Gamma \vdash e : \mathsf{Bool?} \quad \Gamma \vdash e_1 : t? \quad \Gamma \vdash e_2 : t?}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : t?}$$

$$\frac{\Gamma \vdash e : t? \quad \Gamma \vdash e' : t}{\Gamma \vdash \mathsf{try}\, e\, \mathsf{catch}\, e' : t} \qquad \frac{\Gamma \vdash e : t \quad t \text{ not of the form } s?}{\Gamma \vdash \mathsf{try}\, e\, \mathsf{catch}\, e' : t} \qquad \frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \rightarrow t_2}$$

$$\frac{\Gamma \vdash e : s \rightarrow t \quad \Gamma \vdash e' : s \quad s \text{ not of the form } s'?}{\Gamma \vdash e\, e' : t} \qquad \frac{\Gamma \vdash e : (s \rightarrow t?)? \quad \Gamma \vdash e' : s?}{\Gamma \vdash e\, e' : t?}$$

$$\frac{\Gamma \vdash e : s \quad s \leq t}{\Gamma \vdash e : t} \qquad \frac{}{t \leq t?} \qquad \frac{}{t \leq t} \qquad \frac{s \leq t' \quad t' \leq t}{s \leq t} \qquad \frac{s' \leq s \quad t \leq t'}{s \rightarrow t \leq s' \rightarrow t'}$$

Fig. 6. Derived type system for lambda calculus with conditionals and checked exceptions.

This rule covers the case where both e and e' may throw an exception, resulting in an application that may throw an exception. Together with the subtyping rule, this rule also covers the cases where only one of e and e' may throw an exception. Note that the function type $(s \rightarrow t?)?$ has the domain s rather than $s?$ due to the call-by-value semantics: an exception thrown by the argument e' of type $s?$ is propagated as soon as e' is evaluated, which happens before the function is called.

Finally, we transform all derived evaluation typing rules into semantic typing rules using the same argument as in the previous section. From these we then obtain syntactic typing rules by replacing $\models$ with $\vdash$. The full set of rules derived is given in Figure 6. We also include the rules for subtyping, which follow directly from the definition of semantic types.

6 Related Work

To the best of our knowledge, the use of calculational techniques to derive type systems that are sound by construction has not been previously explored in the literature. However, our approach builds upon prior work in a number of areas, as discussed below.

Abstract interpretation. Recently, Garby et al. [2025] proposed a methodology for calculating type checkers. In their approach, a type checker is viewed as a special case of an abstract interpreter [Cousot 1997]. In this view, the correctness of a type checker with respect to a functional semantics is specified as an inequation, which can then be used to calculate the type checker using (in)equational reasoning principles. Similarly to our approach, their calculation approach avoids the explicit use of induction. Instead, the type checker and the evaluation semantics are defined as a fold so that the calculation can exploit a form of fold fusion. However, so far this technique has not been demonstrated to work for higher-order languages such as the lambda calculus. Moreover, the outcome of this technique is a type checker, rather than a set of typing rules.

Semantic type soundness. The notion of semantic type soundness was introduced by Milner along with the notion of type soundness in his seminal paper on polymorphic types [1978]. In this paper, semantic types are assigned to values and used as a way to prove type soundness. However, the underlying proof technique goes back even further to Tait's method to prove strong normalisation for simply typed lambda calculus [1967]. The use of the $\models$ notation to denote the semantic typing relation is due to Appel and McAllester [2001], who used this notation to easily translate typing rules (using $\vdash$) into semantic typing rules (using $\models$) that form the lemmas that need to be proved to show type soundness. We defined the semantic typing relation $\Gamma \models e : t$ in terms of the evaluation typing relation $\gamma \mapsto e : t$, which in turn is based on the definition of semantic types $[\![t]\!]$. While we are not aware of a direct counterpart to the evaluation typing relation in the literature, it is closely related to the notion of a *term relation* (or *expression relation*), and the corresponding generalisation of semantic types to relations is therefore often called a *value relation* [Pitts and Stark 1999].

Categorical logic and type theory. The idea that typing structure emerges from the semantics of a formal language has a long history in type theory and category theory. In type theory, defining types behaviourally by their computational content was pioneered by Martin-Löf [1982], and later further developed in the Nuprl system and its underlying Computational Type Theory (CTT) [Allen et al. 2006; Constable et al. 1986]. Terms in CTT are untyped, and types are defined as specifications of the intended behaviour of such untyped terms. The type system is then constructed on this semantic foundation by essentially proving semantic typing rules. The influential work of Lawvere [1969] showed that reasoning in a category with sufficient structure can be performed using an *internal language* of the category, with structure in the category, e.g. adjunctions, giving rise to syntactic features, e.g. quantifiers, that make up the internal language. This idea has been especially fruitful in topos theory [Johnstone 2002] and type theory [Seely 1984].

7 Conclusion and Further Work

We have demonstrated how to systematically derive sound-by-construction type systems from a specification of type soundness. The key idea is to formulate type soundness semantically and to derive the typing rules as properties of the semantic typing relation. To account for variables in lambda calculi, we have extended our methodology for simple languages described in Section 2.5 with an additional intermediate semantic typing relation $\Mapsto$, which we call evaluation typing. In summary, this generalised methodology proceeds in these five steps:

1. Define a deterministic big-step semantics for the language;
2. Define a pair of semantic relations $\Mapsto$ and $\models$ that capture the desired soundness property;
3. Calculate rules of $\Mapsto$ that have the form (7) of a strictly positive inference rule;
4. Use calculated rules of $\Mapsto$ to calculate strictly positive rules of $\models$;
5. Obtain a sound type system by interpreting the rules of $\models$ as inductive rules for $\vdash$.

This semantic approach not only makes these calculations feasible, but it also makes them modular. We can calculate semantic typing rules one language feature at a time, and in some cases we can reuse semantic typing rules across different languages with shared language features. This modular approach also carries over to our Agda formalisation of the results.

The semantic approach to type soundness has seen a significant resurgence in recent years and has produced a rich body of work, including the development of Kripke logical relations to prove soundness for languages with various forms of effects, and the advancement of formalisation techniques for mechanising such soundness proofs [Abel et al. 2019]. We believe that by drawing on this abundance of semantic reasoning techniques, future work can extend our methodology of calculating sound-by-construction type systems to a wider range of language features such as non-termination, non-determinism, effectful computations, and polymorphic types. For example, in preliminary work we have explored how non-termination can be handled using the partiality monad, and non-determinism by using a more general form of type soundness.

Acknowledgements

We would like to thank Brandon Hewer for useful comments and suggestions about our methodology. 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

Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark Reloaded: Mechanizing Proofs by Logical Relations. *Journal of Functional Programming* 29 (2019).

S. F. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. 2006. Innovations in Computational Type Theory Using Nuprl. *Journal of Applied Logic* 4, 4 (2006).

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Transactions on Programming Languages and Systems* 23, 5 (2001).

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

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc.

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

Zac Garby, Patrick Bahr, and Graham Hutton. 2025. The Calculated Typer. In *Proceedings of the ACM SIGPLAN International Haskell Symposium.*

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

Peter T Johnstone. 2002. *Sketches of an Elephant: A Topos Theory Compendium.* Oxford University Press.

F. William Lawvere. 1969. Adjointness in Foundations. *Dialectica* 23 (1969).

Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. *Studies in Logic and the Foundations of Mathematics* 104 (1982).

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978).

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

A. M. Pitts and I. D. B. Stark. 1999. Operational Reasoning for Functions with Local State. In *Higher Order Operational Techniques in Semantics.* Cambridge University Press.

R. a. G. Seely. 1984. Locally Cartesian Closed Categories and Type Theory. *Mathematical Proceedings of the Cambridge Philosophical Society* 95, 1 (1984).

W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite type I. *The Journal of Symbolic Logic* 32, 2 (1967).

Alfred Tarski. 1955. A Lattice-Theoretical Fixpoint Theorem and its Applications. (1955).

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, Article 40 (2024). Issue 6.

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994).