Imperial College London
Department of Computing

Context-Oriented Functional Programming

Pedro M. Martins

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College London

Abstract

The modern computing landscape, featuring heterogeneous interconnected mobile devices, poses new challenges and opportunities for application development. Mobility and heterogeneity of devices imply that applications need to adapt depending on their execution context. Contexts such as the device that an application is running on, or the power profile, may require widespread program-wide adaptation. Dealing with this adaptation can lead to the introduction of subtle bugs, and subsequent runtime errors, such as null pointer exceptions when context has not been initialized. Current approaches to encoding context-aware behaviour are either very flexible but unsafe, or safe but too restrictive. In this thesis we present a new approach to context-aware application development based on functional programming, which attempts to be both flexible and safe. In order to do so, we present an embedded domain specific language in Haskell, where we explore the design space of context-dependent values within a functional programming language. In particular, we explore how to use Haskell's type system to automatically derive the context dependencies needed by a computation at the type level, and use this to ensure that required context is always available.

We then develop *context-dependent types* to ensure safety in the presence of program-wide variation. By using different return types for different modes of operation of the program when appropriate, we can ensure isolation between them through type checking. We extend our domain specific language to support context-dependent types, whilst retaining type soundness, as well as sound and (we conjecture) complete type inference. We present a core calculus for these features and a high-level language that extends the calculus with practical programming features. Evaluation is performed by examining a context-aware application requiring exactly the kind of adaptation that is unsafe to implement in current approaches. We show that our language compares favourably to the state of the art in terms of both safety and code clarity.

The work presented in this thesis has been funded by Fundação para a Ciência e a Tecnologia (Portugal) under grant SFRH/BD/61917/2009.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank:

- My supervisors, Dr. Julie McCann and Prof. Susan Eisenbach for their continuous encouragement, enthusiasm, support and guidance. Through their great experience and knowledge, they helped me not only develop the contributions that I will present, but also learn how to present them and motivate them, and interest other people in them.

- Prof. Sophia Drossopoulou, for her endless enthusiasm and patience in discussing preliminary drafts and unfinished ideas and helping motivate and develop the systems in this thesis.

- The SLURP and AESE groups at Imperial College, for various interesting discussions and feedback on my research. Especially, and in no particular order, I have to thank Dr. Tristan Allwood, Sylvan Clebsch, Dr. Will Jones, Roman Kolcun, Dr. Reuben Rowe and Tim Wood for their enthusiasm in brainstorming the concepts in this thesis and several improvement suggestions.

- The Fundação para a Ciência e a Tecnologia for providing me with the grant that allowed me to pursue this PhD.

- My family and friends, for their encouragement, love and support, and for ultimately setting me on the path that led me here.

CHAPTER 1

Introduction

1.1 Motivation and Objectives

Computing lies at the core of modern services; from smart vehicles, smart medicine, smart buildings, and even smart cities. Key to this 'smartness' is the design of flexible services, tailored to the users' needs and that make sense given the time and place that the user is currently in, and taking into account their preferences. The information that drives this ability to tailor and adapt dynamically is known as context [SDA99]. This contextual information ranges from immediately available data such as the device, time of day and location of the user, to more complex derived information such as the user's social networks, and their friends' locations. Context can be obtained from user profiles that have been gathered manually or even via crowd-sourcing, it can be determined using tiny sensing devices or from sensors in one's phone. This idea has extended to computer architectures that also adapt dynamically, tailoring their behaviours in a way that is again driven by context; which can be user related or directly reflecting the environment or conditions that the device finds itself in.

One example of the latter, where the operation of a system is determined by the perception of its current environment can be found in laptops. Laptops are typically battery powered and while untethered they are required to maximise battery lifetime. One way to do this is by providing two graphics cards; one integrated and the other external. For non-graphics intensive applications, one can take advantage of the integrated card to preserve battery power; alternatively the external card is then used for more graphics-intensive applications. If one were to implement this, then the behaviour of the applications that use the cards must change depending on the graphics subsystem being used at that moment of time. However these subsystems may be radically different in terms of presenting and returning different

input/output types. For example, the external card operates with precise internal values of double precision while the internal card uses single-precision floating point values. This change would have to permeate throughout the whole system.

Essentially this forces developers to encode variability at a global level, in a dynamic way, so that the mode of operation is decided depending on the current sensed context. We call these different versions of the code *global alternatives*. These notions of global variability are incompatible with the abstraction and modularity assumptions which typify practical programming languages. In fact, what we observe is that current abstraction mechanisms prevent reusability when variability is present at a global level, and attempts at abstracting this via frameworks, such as Context Toolkit or JCAF, have typically required the developer to restructure the application completely to integrate the proposed framework. This is because representation changes are required to map the applications' data to the framework's representation of the data, and framework specific commands are required to allow the application to use the framework. Moreover, a runtime system is required to manage this, which invalidates abstraction assumptions in the host language. This incompatibility typically manifests itself as ad-hoc type systems that are only checked at runtime and are not properly integrated with the type system of the programming language. This can lead to crashes during the execution of programs written in these languages, even when the error is evident from the static properties of the program. For example, assuming that a memory pointer points to a value with a different representation than what it has can lead to null pointer dereferences and subsequent runtime crashes.

The importance and role of context in modern computing has been long foreseen [Wei95] and within the Pervasive Computing community prototypical implementations of context-aware applications, as well as frameworks to deal with context, have been proposed. With widespread availability of mobile computing devices such as mobile phones and tablets, practical implementations of context-aware applications have started to appear. Indeed, applications for mobile phones have started to incorporate location awareness ubiquitously, in order to provide the user with more relevant information. For example, there are event applications which will show the events that are closest to the user first or reminder applications that will only get activated at certain times, in certain locations. However, we are witnessing a divide between the solutions proposed by researchers and the practical solutions adopted by implementers. We believe that this is because the former solutions are too heavyweight and rigid, and force developers to re-engineer their applications dramatically to incorporate them. As a result, practical implementations are typically based on bespoke implementations of context-aware behaviour, which on one hand prevents reusability of behaviour, and on the other hand makes it easier for subtle bugs and programming errors to

be repeated throughout implementations of the same behaviour. These difficulties make it harder to explore richer contextual dependencies, or to build on previously implemented context-aware behaviours. It has been argued that this abstraction tradeoff is an inevitable consequence of context-awareness. Indeed, Lieberman and Selker [LS00] present a simple model for context-awareness and postulate that due to the dynamic nature of context-aware applications, it is hard to specify a module's behaviour in a way that will allow it to be reused at all, with current abstraction techniques. We believe that through a deeper embedding of context-awareness semantics into a programming language, we should be able to specify some of this behaviour and provide natural programming language constructs for it. In addition to this, by being aware of the semantics of context-awareness, a compiler for such a language should be able to verify statically whether certain properties that we believe should be true for this type of behaviour actually hold. For instance, we can check that the developer provides a given type of contextual information by querying the relevant sensors before it is required, or we can check that values from one context are not used in another when that does not make sense from a typing point of view.

Previous type-sound approaches that could be applied to solving the context-awareness abstraction problem include context-oriented programming [SGP11] and dependent types [AMM05]. As we will see, while context-oriented programming can be used for solving the global variability problem, it is unable to cope with return types changing depending on context, as would happen in the graphics card example outlined above. Dependently-typed systems can encode the full variability that is needed in theory, however, they are lacking in practical applicability [McB02]. For instance, restrictions to the forms of recursion that are allowed must be in place in order to ensure decidability of type checking, as arbitrary expressions are allowed at the type level. Complete type inference is also undecidable in the presence of unrestricted dependent types [Bar92]. New dependently-typed languages, such as Idris [Bra13], have made dependent types more practical, by reducing the impact of these fundamental limitations. However, the limitations are still present and affect the practicality of dependently typed languages. Moreover, we are unaware of any approach within a dependently-typed system that tries to create abstractions for naturally expressing dynamic adaptability based on globally accessible information. This presents us with two options. We can either attempt to mitigate the impact of the limitations of dependently-typed languages so they have a minimal effect on practical development of context-aware adaptation, or we can develop a language with a subset of dependent types that is targeted to our application and thus does not pose those limitations. We choose the latter in this thesis.

We think that by exploring the connection between context and applications, through providing a faithful model for context and how it affects programs, we can design libraries that

allow developers to integrate context easily into their applications, and have more soundness guarantees than before. We can thus eliminate whole classes of bugs, such as mixing values from different contexts that cannot be mixed. This could allow developers to create more safely and manageably much more complex types of context-aware behaviour than we have seen to date.

Based on the problem described and the current state of the art, the design goals for the programming principles we want to design in this project are as follows (the parenthesized denominations will be used as abbreviations for these design goals throughout the thesis):

- Lightweight context usage (*Light*): Context usage should be lightweight in terms of syntax and abstractions, and provision of context should be deferred to a runtime or a top level context loop in the program. Ideally the programmer should be able to refer to contextual values merely by their identifier, for instance "UserLocation", and not have to restructure the enclosing module to allow it to depend on contextual information. For example, we should be able to change a non-context dependent distance calculation with regards to some reference point, by making it use the current location of the device:

  ```
  dist = distance reference venue
  dist' = distance ?(User,IsLocatedAt) venue
  ```

- Composability (*Comp*): Context-dependent computations should behave as regular values, so we should be able to apply functions to them and otherwise manipulate them. Composability is aligned with lightweight context usage. If we make a module context-aware by merely referring to a contextual value, we ought to not have to change the code of the users of that module. The type of the module can change provided that the type system is made flexible enough so that the resulting program is well-typed. For instance:

  ```
  dist + 1
  dist' + 1
  ```

  Both the above expressions should be correct.

- Separate reusable definition of ontology/typing/context runtime (*Ont*): The programmer should be free to define, in a separate module, either an ontology or a context runtime. By ontology we mean a definition of the domain of contextual values we are interested in and what types these contextual values will have. The context runtime is then a collection of procedures for retrieving the defined types of contextual values using a particular devices' sensors. It should be possible to specify these separately from the application code to maximise modularity and reusability of these definitions.

In our example, we should be able to define that UserLocation stands for a contextual value, of type Location, and how to retrieve it for the particular device we are using. In a language that will be defined in Chapter 5 the ontology takes the following form:

```
individual User
feature IsLocatedAt :: Location
relevant Location (User ▷ IsLocatedAt) by distance
```

The context runtime is then a set of definitions of how to retrieve a particular type of contextual information (in this case, for instance, querying the location sources of the phone, potentially using redundancy to maximize accuracy):

```
instance Realizable (User ▷ IsLocatedAt) where
...
```

- Context-dependent types (*CDT*): The return type of the context-dependent dispatch procedures need not be the same. This has important consequences for composability. It is the type system's responsibility to ensure that all the necessary alternatives are provided and that they are compatible. For an example of what is meant by context-dependent types, consider the following simplified example from Chapter 7, whereby depending on the value of the contextual `protocol` (`P1` or `P2`) we want to return a different type (`Status1` or `Status2` respectively). We use the **ccase** construct to introduce a modal branch depending on the runtime value of the `protocol` contextual value:

```
alert : Case protocol of
  { P1 ⇒ Status1
  ; P2 ⇒ Status2}
alert = ccase protocol of {
  P1 ⇒ S1;
  P2 ⇒ S2;
}
```

The type of `alert` reflects this modal contextual dependency. This adaptation is the type of behaviour that would typically be solved by casting to a common supertype of `Status1` and `Status2`, paired with a dangerous downcast when this value is used. With our system we can remain type safe even in the usage sites:

```
process : Case protocol of { P1 ⇒ Status1 → IO ()
                           ; P2 ⇒ Status2 → IO () }
process s = ccase protocol of {
  P1 ⇒ .. S1 ..
  P2 ⇒ .. S2 ..
```

```
}

main = process alert
```

This example is already a compelling use case of context-dependent types, and will be further examined in detail in Chapter 7.

- Strong typing and Safety (*Safe*): Type soundness with regards to the defined ontology even in the presence of context-dependent types. Type soundness allows us to regain reasoned composability, despite the flexibility provided by lightweight context usage and context-dependent types. In the previous example, returning something of the wrong type for the status would not type-check:

```
-- Type error!
alert : Case protocol of
  { P1 ⇒ Status1
  ; P2 ⇒ Status2}
alert = ccase protocol of {
  P1 ⇒ S1;
  P2 ⇒ S1;
}

process : Case protocol of { P1 ⇒ Status1 → IO ()
                           ; P2 ⇒ Status2 → IO () }
process s = ccase protocol of {
  P1 ⇒ .. S1 ..
  P2 ⇒ .. S2 ..
}

main = process alert
```

- Type inference (*Inf*): Since the types may get fairly complex when keeping track of all this information, type inference is essential for practical applicability. Moreover, the system should be as permissive as possible in not mandating any artificial ordering or nesting of contextual program flow branches either through canonical forms or equivalences. For instance, in the previous example, the types of `alert` and `process` should be inferrable.

1.2 Contributions

Our contributions are as follows:

- A composable abstract model for context-dependent values, based on functional programming. We show how functional programming can be used to model naturally some concepts in context-awareness in a well-reasoned way. We then develop two prototypical formal systems that feature this model, one suitable for exploring the embedding of the model in existing functional programming languages, and the other to further explore the metatheory associated with the model.
- An encoding of this model in a domain-specific language that embodies the ideas of context-dependent values and context sources (HCONTEXT). The language features an embedded type system that ensures type soundness in the presence of flexible context-awareness constructs. We then use this language to explore the design space of context representation and examine the interaction between the representation choice and our abstractions.
- A formal grounding for context-dependent types in a functional programming setting in the form of a type-sound calculus, λ_{env}, as well as mechanically verified proofs of type soundness for it. This calculus also features a type inference algorithm that is sound and (we conjecture) complete, and we present mechanically verified proofs of those properties.
- A proposal for a practical implementation of these ideas, presented through a high-level language containing the concepts from the previous two explorations (CDT). Using this language we provide an example application and reflect upon the advantages of our approach.

1.3 Statement of Originality

Except where otherwise referenced, all contributions in this thesis are my own.

1.4 Publications

Along the process of performing the research that is presented in this thesis, I have published papers detailing case studies and contributions:

- *Pedro M. Martins, Sophia Drossopoulou, Susan Eisenbach and Julie A. McCann. Type-Safe Global Environments and Alternatives. Journal of Functional Programming (under review)*:

  This paper describes the calculus for context-dependent types which will be expounded in Chapter 6. It is currently under review.

- *Pedro M. Martins, Julie A. McCann, and Susan Eisenbach. The Environment as an Argument: Context-Aware Functional Programming. In Proceedings of the 14th international symposium on Practical Aspects of Declarative Languages, PADL'12, pages 48–62, Berlin, Heidelberg, 2012. Springer-Verlag*:

  This publication describes the domain-specific language that will be described in Chapter 5. This chapter is heavily based on the paper.

- *Pedro Martins and Julie A. McCann. ajME: Making Game Engines Autonomic. In Proceedings of the 3rd International Conference on Fun and Games, Fun and Games '10, pages 48–57, New York, NY, USA, 2010. ACM*:

  This paper carries over from previous work, but is tangentially related to the motivating problem, as it presents context-aware adaptation within a gaming engine. It uses the autonomic framework [IBM06] proposed by IBM.

- *Michael Breza, Pedro Martins, Julie A. McCann, Evangelos Spyrou, Poonam Yadav, and Shusen Yang. Simple solutions for the second decade of wireless sensor networking. In Proceedings of the 2010 ACM-BCS Visions of Computer Science Conference, ACM-BCS '10, pages 7:1–7:12, Swinton, UK, UK, 2010. British Computer Society*:

  This paper was co-written with the AESE group, and presents an examination of the current state of wireless sensor network research, as well as some suggestions for the future. It is not immediately relevant to this thesis, but it is nevertheless an interesting case study of a field that is related to Pervasive Computing, and the Internet of Things, where the issues in this thesis have since become more and more relevant [PB08].

1.5 Structure of the thesis

Given that we are going to introduce several systems throughout the thesis it is useful to visualise how they extend each other, and in which ways they are similar and distinct. The context-awareness definition and formalisation is going to begin with $\lambda_{\downarrow}$ (Chapter 4), a skeleton system that embeds the essence of context-awareness in a functional programming language. We use it to discuss what the effects of context-awareness on types should be. We

then embed this system in Haskell [M+10] in order to explore the design space of context-dependent values and context representations within a strongly typed functional programming setting. This results in a Haskell-based DSL called HCONTEXT (Chapter 5). In a parallel track we then extend $\lambda_{\downarrow}$ with context-dependent types in a core calculus which we name λ_{env} (Chapter 6). We then integrate the ideas from HCONTEXT in the λ_{env} core calculus in order to form a standalone practical programming language which we call CDT (Chapter 7). This is summarised in the following diagram:

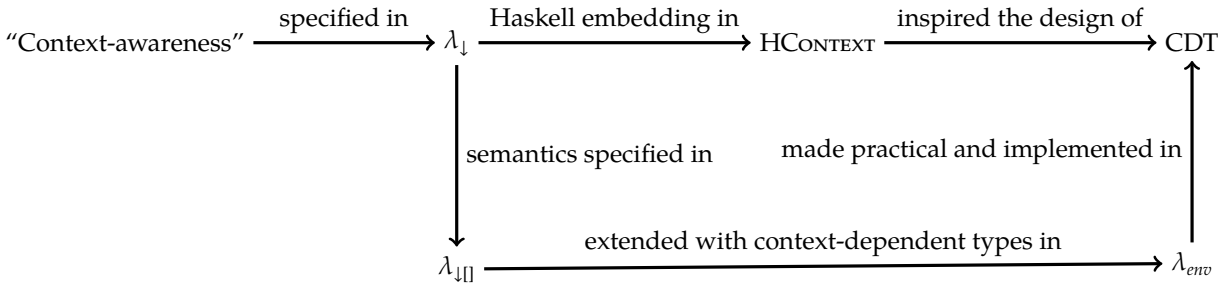

Figure 1.1: Structure of the thesis

1.6 Notation

Throughout the thesis, we will use overline notation for sequences, as has become usual practice in programming language research and denote sequence concatenation by ; and the empty sequence as $\bullet$. This notation extends to sequences of more complex structures in an unusual way, as in the following example:

$$\overline{a \Rightarrow b} \quad \text{stands for} \quad a_1 \Rightarrow b_1; ..; a_n \Rightarrow b_n$$

This choice is deliberate, to e.g. make it clear when we are talking about a sequence of pairs or a pair of sequences. In the cases where a variable already has a subscript, this notation will simply append the enumerated subscript in the following way:

$$\overline{a_1 \Rightarrow a_2} \quad \text{stands for} \quad a_{1,1} \Rightarrow a_{2,1}; ..; a_{1,n} \Rightarrow a_{2,n}$$

We also use $\#(\overline{a})$ to stand for the length of sequence $\overline{a}$. It will be useful in certain cases to have certain portions of a sequence remain constant. We will use explicit indices in those cases:

$$\overline{a \Rightarrow b_i}^{\,i} \quad \text{stands for} \quad a \Rightarrow b_1; ..; a \Rightarrow b_n$$

We will use sequence notation in formulas within inference rules to denote the conjunction of all the formulas in the sequence. Whenever there is no scope for ambiguity, we may also treat sequences as sets of the elements in the sequence. Otherwise, this will be denoted as $\{\overline{a}\}$.

Moreover, whenever we state properties, all variables are to be interpreted as being universally quantified unless said otherwise.

If a metavariable x ranges over a set, by convention the primed metavariable x' will also range over the same set, as well as a subscripted one, e.g. x_y. Lowercase subscripts range over indices, while uppercase subscripts merely denote labels for the variable. So for instance x_1 and x_Y are different metavariables, whereas x_y can be instantiated to be x_1.

1.7 Automation

Most of the metatheory presented in this thesis has been validated using the Ott tool [SNO+07], and proven using the Coq proof assistant [Coq02]. The commented proof scripts for the main results are presented in full in Appendix B, which also presents an overview of the embeddings used for all the mathematical objects within the logic used by Coq. All the definitions presented using inference rules are thus TEX output, produced by Ott. The auxiliary definitions are presented in traditional mathematical notation, but the Coq encoding can be found in the appendix. This allows us to ensure a higher level of certainty in the results presented. The Haskell definitions in chapter 5 are valid GHC Haskell, modulo additional syntactic sugar which will be introduced and fully defined. The relevant desugared source listings in GHC Haskell are presented in full in Appendix C.

CHAPTER 2

Background

Throughout this thesis we will use the lambda calculus as a formal system in which we represent our notions of computation. Moreover, in chapter 5, we will use the functional programming language Haskell [M+10] in order to embed some of the abstractions. In order to make the presentation in this thesis self-contained, we present in this chapter an overview of the lambda calculus, as well as some of the specificities of Haskell with regards to other functional programming languages.

2.1 Lambda Calculus

The lambda calculus is a model of computation that is based solely around the notions of function abstraction and function application. Despite its simplicity it is able to express all computable functions [Bar92]. This led to its adoption as a formal model for reasoning about programming languages.

Definition 2.1. *The set of λ-terms, ranged over by $e, e', ...$ is built from an infinite set of variables (ranged over by $x, y, z, ...$, using application (denoted by e e) and (function) abstraction (denoted by $\lambda x.e$):*

$$e \in Expr ::= x \mid e\ e \mid \lambda x.e$$

We will abbreviate iterated application by assuming that it is left-associative, e.g. $ee'e''$ stands for $(e\ e')\ e''$, and iterated abstraction, by allowing multiple variables in one abstraction term, e.g. $\lambda xy.e$ stands for $\lambda x.\lambda y.e$. In the term $y\ (\lambda x.e)$ the variable x is said to be *bound*, and the variable y is said to be *free*. The notation $[x := e]e'$ denotes the capture-avoiding substitution

of all the free occurences of x in e' by e. During this process, should the free variables of e become bound, the free variables of e and the bound variables of e' should be renamed to avoid capture. This will happen silently on paper, but when developing an implementation of the lambda calculus, other binding representations for binding are useful, such as DeBruijn indices [Bru72]. Reduction is then defined through a relation $\rightarrow_\beta$ which is defined in Figure 2.1.

$\boxed{e \rightarrow_\beta e'}$

$$\frac{}{(\lambda x.e)\, e' \rightarrow_\beta [x := e']e} \quad \textsc{Beta}$$

$$\frac{e_1 \rightarrow_\beta e_2}{e\, e_1 \rightarrow_\beta e\, e_2} \quad \textsc{AppArg}$$

$$\frac{e_1 \rightarrow_\beta e_2}{e_1\, e \rightarrow_\beta e_2\, e} \quad \textsc{AppFun}$$

$$\frac{e \rightarrow_\beta e'}{\lambda x.e \rightarrow_\beta \lambda x.e'} \quad \textsc{LamCong}$$

Figure 2.1: Reduction for the lambda calculus

$\twoheadrightarrow_\beta$ is then defined as the transitive-reflexive closure of $\rightarrow_\beta$ and it has the fundamental property that it is confluent:

Theorem 2.1. *Confluence (Church-Rosser):*

$$e \twoheadrightarrow_\beta e' \wedge e \twoheadrightarrow_\beta e'' \Rightarrow \exists e'''.e' \twoheadrightarrow_\beta e''' \wedge e'' \twoheadrightarrow_\beta e'''$$

The proof of this theorem is standard, and can be found for instance in [Bar92].

Type systems for the lambda calculus assign types to terms that allow us to verify that application terms are well-formed, and will reduce. This is encoded as a type assignment system.

Definition 2.2. *The set of types for the simply-typed lambda calculus is generated from an infinite set of type variables (ranged over by $X, Y, Z, ...$) as follows:*

$$t \in Ty ::= X \mid t \rightarrow t$$

Moreover, we abbreviate iterated application of $\rightarrow$, by assuming it is right-associative, e.g. $t \rightarrow t' \rightarrow t''$ stands for $t \rightarrow (t' \rightarrow t'')$. In order to capture assignments of types to bound variables, an environment is used, ranged over by Γ, which can be thought as a partial function

from variables to types. The type assignment system then consists of judgements of the form $\Gamma \vdash e : t$, which is read as: under environment Γ, term e has type t. If there is a type for a term e such that $\Gamma \vdash e : t$, the term e is said to be well-typed. The rules for this type system are defined in Figure 2.2. The Ax rule merely uses the environment to look up the type of a variable. The Lam rule is used to type lambda abstractions, and uses an environment extended with the typing for the variable bound in order to type the body of the lambda. App then types function application, by enforcing the argument type of the function and the type of the argument to match, and typing the application term with the return type of the function.

$\boxed{\Gamma \vdash e : t}$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t}\ \textsc{Ax}$$

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x.e : t \to t'}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash e\, e' : t'}\ \textsc{App}$$

Figure 2.2: Type system for the simply-typed lambda calculus.

Type soundness is then informally the property that if we can assign a type to a term, then we can either eventually reduce it to a normal form or reduction will not terminate, but we will never reach a stuck state. Normal forms constitute the final result of a computation. For example, we can say that we are interested in terms of the form $\lambda x.e$. This is called weak head normal form, and is defined as:

$$v \in Val ::= \lambda x.e$$

Type soundness is commonly shown in two steps, progress and type preservation [WF92]:

Theorem 2.2. *Progress: Given a closed term e, if e is typeable, then it is either a value or is further reducible:*

$$\vdash e : t \Rightarrow e \in Val \vee \exists e'.e \to_\beta e'$$

Theorem 2.3. *Type Preservation: If a term e is typeable, and we reduce it one step, the resulting term is also typeable, with the same type:*

$$\Gamma \vdash e : t \wedge e \to_\beta e' \Rightarrow \Gamma \vdash e' : t$$

Proofs of both these theorems are by standard induction over derivations and can be found for instance in Pierce's book [Pie02]. Moreover, all typeable terms in the simply typed lambda calculus are strongly normalizing, i.e. all reduction sequences starting with a typeable term terminate. The proof is also standard and can be found for instance in Barendregt's typed lambda calculi survey chapter [Bar92].

2.1.1 Parametric Polymorphism

The lambda calculus can be extended to allow a function to be written generically, in a way that allows it to be used over any type. For instance, the identity function $\lambda x.x$ can be assigned the type $t \to t$, for arbitrary $t \in Ty$. This can be encoded in the type system, by extending the syntax of types to include universal quantifiers for type variables. There are two approaches to this, which are called Church-style and Curry-style in the literature, depending on whether we modify the term-level syntax or not. In this thesis we will focus on Curry-style systems, where we do not. We can thus define type schemes as containing universally quantified type variables[1]:

$$\sigma ::= t \mid \forall X.\sigma$$

Using these type-schemes, we can then assign a more generic type to the identity function:

$$\lambda x.x : (\forall X.X \to X)$$

Universally quantified type variables are said to be bound in the scope of the quantifier. We denote the set of type variables and free type variables of σ as $tv(\sigma)$ and $ftv(\sigma)$ respectively. These operations extend to environments in the obvious way, as the union of the (free) type variables of all the type schemes contained within. The type system judgements can then be modified to include generalisation and instantiation of type variables, as is shown in Figure 2.3.

We can prove equivalent progress, type preservation and strong normalization theorems for the polymorphic lambda calculus [Bar92]. Damas and Milner [DM82] proved that there is a sound and complete type inference algorithm for this calculus, which they called algorithm W. This algorithm is predicated on the existence of a unification algorithm $\mathscr{U}$.

[1] The stratification of types into type schemes and regular types is necessary to ensure decidable type inference [Bar92]. Unrestricted polymorphic systems, such as system F [GTL89] have very interesting properties, such as being able to incorporate Church encodings to embed the natural numbers [GTL89]. However, in this thesis we will focus on Hindley-Milner style polymorphism to ensure decidable inference.

$\boxed{\Gamma \vdash e : \sigma}$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \textsc{Ax}$$

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x.e : t \to t'}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash e\, e' : t'}\ \textsc{App}$$

$$\frac{\Gamma \vdash e : \forall X.\sigma}{\Gamma \vdash e : [X := t]\sigma}\ \textsc{Inst}$$

$$\frac{\Gamma \vdash e : \sigma \quad X \notin ftv(\Gamma)}{\Gamma \vdash e : \forall X.\sigma}\ \textsc{Gen}$$

Figure 2.3: Type system for the polymorphic lambda calculus

Definition 2.3. *There is an algorithm $\mathscr{U}$ which given two types t and t', either fails or returns a substitution S of type variables for types such that:*

- $S(t) = t'$
- *S is the most general unifier, i.e. if another substitution S' unifies t and t', there is another substitution S'' such that $S' = S''S$.*
- *S involves only variables in t and t'.*

Moreover, we order types according to a relation $\preceq$, reflecting instantiation, which is defined as follows:

$$\forall \overline{X}.\, t \preceq \forall \overline{X'}.t' \overset{def}{\iff} t' = [\overline{t/X}]t \wedge \overline{X'} \cap ftv(\forall \overline{X}.t) = \varnothing$$

Our presentation of algorithm W, in Figure 2.4, differs from the original slightly, in that we present it as inference rules. We should note that this is a slight abuse of notation as it relies on side effects for providing fresh variables. One way to model this formally is to thread an infinite tape of fresh variables through the algorithm, as can be seen in [Vau08]. We will adopt that approach in the mechanized proofs of our results. The original presentation also modified the syntax of terms by adding a *let* expression, allowing types with polymorphic terms to be locally introduced and added to the environment. We omit this here for simplicity,

as we are focusing on type soundness and type inference in the presence of polymorphic types in the environment. Adding these types to the environment could be achieved, for instance, by designing a higher-level language where we perform type inference on all top-level declarations and generalise the types before adding them to the environment, which is in effect what the typing rules for *let* do.

$\boxed{W(\Gamma, e) = (S, t)}$

$$\frac{\begin{array}{c} x : \forall \overline{X}.t \in \Gamma \\ \overline{X}' \textbf{ fresh} \end{array}}{W(\Gamma, x) = (\mathbf{id}, [\overline{X := X'}]t)} \text{ IAx}$$

$$\frac{\begin{array}{l} W(\Gamma, e_1) = (S_1, t_1) \\ W(S_1(\Gamma), e_2) = (S_2, t_2) \\ S = U(S_2(t_1), t_2 \to X) \\ X \textbf{ fresh} \end{array}}{W(\Gamma, e_1\, e_2) = (S \circ S_2 \circ S_1, S(X))} \text{ IApp}$$

$$\frac{\begin{array}{c} W(\Gamma, x : X, e) = (S, t) \\ X \textbf{ fresh} \end{array}}{W(\Gamma, \lambda x.e) = (S, S(X) \to t)} \text{ IAbs}$$

Figure 2.4: Type inference algorithm for the polymorphic lambda calculus

In order to state completeness, we also need to define closure of a type t with respect to Γ:

$$\overline{\Gamma}(t) = \forall \overline{X}.t, \overline{X} = ftv(t) \setminus tv(\Gamma)$$

Soundness and completeness are then stated as follows:

Theorem 2.4. *Soundness: If $W(\Gamma, e)$ succeeds with (S, t) then there is a derivation of $S\Gamma \vdash e : t$:*

$$W(\Gamma, e) = (S, t) \Rightarrow S\Gamma \vdash e : t$$

Theorem 2.5. *Completeness: If $\Gamma \vdash e : \sigma$ then $W(\Gamma, e)$ succeeds with $W(\Gamma, e) = (S, t)$, where $\overline{S\Gamma}(t) \preceq \sigma$:*

$$\Gamma \vdash e : \sigma \Rightarrow W(\Gamma, e) = (S, t) \wedge \overline{S\Gamma}(t) \preceq \sigma$$

Proofs of these theorems can be found in [Dam84].

2.1.2 Recursion and Data Types

In order to extend the aforementioned calculi into a practical programming language we then need to encode features such as recursion and data types. Recursion can be added by defining a polymorphic fixed-point operator fix. We can add the typing $fix : \forall X.(X \to X) \to X$ to the environment Γ and the reduction rule: $fix\ f \ \to_{fix}\ f\ (fix f)$. Obviously with this rule, we lose the strong normalization property, but the system is still type-sound. The encoding of recursion as fixed-points is routine [Bar92]. Numbers and other data types can also be defined as primitives in the polymorphic lambda calculus by small extensions to the type system/operational semantics that preserve all the properties we discussed so far [Pie02].

2.2 Haskell

Haskell has emerged as an experimental playground for trying out alternative type systems [HHJW07]. This is partly because of its flexible type class mechanism which allows the programmer to include certain constraints in the type system, thus allowing such type systems/languages to be embedded as domain specific languages.

Type classes were introduced by Philip Wadler [HHPJW96], originally to extend the Hindley-Milner type system to add overloading of arithmetic operators to different numeric types. Type classes specify operations that must be implemented for a type in order for it to be a member of that class (called *instance*). For example, a type class `Eq` that specifies that values of a type can be compared for equality, specifies an operation that allows equality comparison of two values of that type. This is specified in Haskell as:

```
class Eq a where
   (==) :: a -> a -> Bool
```

This effectively defines a set of types that can be compared for equality. To say that a type T is an instance of that class is then specified as:

```
instance Eq T where
   (==) = ...
```

In order for the type class mechanism to interact with polymorphism, Haskell adds constraints to types, forming predicated types. We can then constrain a type variable α to be a

member of a type class κ with the constraint $(\kappa\,\alpha)$. The full syntax for types is then as follows, where κ is a type class, τ a type and α a type variable:

$$\sigma ::= \forall\alpha_1 \ldots \alpha_l.\langle\kappa_1\,\alpha_1, \ldots, \kappa_m\,\alpha_m\rangle \Rightarrow \tau$$

This allows us to specify a polymorphic type for `(==)`, which is rendered in Haskell as:

```
(==) :: (Eq a) => a -> a -> Bool
```

Programs using type classes are then transformed into programs in a simpler language that may be typed using the Hindley-Milner rules, so the same algorithm for inference can be used. This is shown in the following example, where the type could have been left out and inferred by the compiler:

```
tripleEq :: (Eq a) => a -> a -> a -> Bool
tripleEq x y z = x == y && y == z
```

This functionality is integrated as part of the Haskell 98 standard [Jon03] along with a mechanism that allows automatic derivation of type class instances for algebraic data types (e.g, deriving `Eq` instances for the algebraic data type `data T = A | B`).

In some implementations of Haskell, like GHC, this functionality has then been further extended to allow type classes to take multiple parameters. This can be used to model relations between types[2]. For a valid example, consider the following class:

```
class T a b where
   fn :: a -> b
```

GHC by default enforces several restrictions to ensure decidable type checking in the presence of type classes with multiple parameters. These restrictions are analysed in [JJM97], corresponding, for example, to restrictions on the form of context (the type class restrictions on types), how much context reduction should be done before generalising a type and whether to allow overlapping instances. GHC's design of the `MultiParamTypeClasses` extension demands that the context of a class declaration must be simple, i.e consist of classes applied to type variables, and that the class hierarchy be acyclic. For example, this declaration is not accepted (and in fact would not be if `T` were a single-parameter type class either):

[2]Two types a,b are in a relation $R(A, B)$ if there is an `instance R a b`

```
class Eq [a] => T a b where
   ...
```

In addition to this, instance declarations have to be of the form

```
instance (assertion1, ..., assertionn) => class x1... xm where ...
```

where `x1..xm` are type variables. Assertions must be of the form `C x1 ... xm`. This forbids instance declarations such as:

```
instance C (Maybe Int) where ...
```

These restrictions are added in order to retain decidable type checking and separate compilation.

This seemingly simple addition, however, brings with it many opportunities for ambiguity. The classical example is that encountered when defining a library of collection types [Jon00]. In defining such a library one might want to define a type class for collections:

```
class Collects e ce where
  empty  :: ce
  insert :: e -> ce -> ce
```

This type class defines a set of functions that one could reasonably expect from a collection. For instance, we could define an instance for lists:

```
instance Collects a [a] where
  empty = []
  insert = (:)
```

However, empty now has an ambiguous type:

```
empty :: Collects e ce => ce
```

This type is ambiguous as there could be more than one alternative for the type variable `e`. For instance when evaluating the term `empty :: [Int]`, there is no way for the compiler to know that there is only one meaningful value for `e` in `Collects e ce`, and thus this term is not well defined. GHC extends typeclasses with syntax to specify exactly that dependency. We can specify the additional functional dependencies in a type class with the syntax `var1 .. varm -> var'`, which states that the sequence `var1 .. varm` uniquely determines `var'`. For instance in the Collects example, we can add the functional dependencies to the class head, separated by a '`|`':

```
class Collects e ce | ce -> e where
...
```

We state that `ce` uniquely determines `e`. Thus, when evaluating the term `empty :: [Int]`, the compiler knows that there can only be one instance for `Collects e [Int]`, and the type `Collects e [Int] => [Int]` is no longer ambiguous.

GHC also allows for the restrictions on instance heads to be lifted in a controlled manner. GHC provides a set of extensions for that purpose which can be enabled by the programmer on a per-module or per-program basis:

FlexibleInstances: The type class can be applied to arbitrary types in the instance head (not just type variables). For instance, the previous invalid instance would now be allowed:

```
instance C (Maybe Int) where ...
```

FlexibleContexts: Assertions in contexts can contain type classes applied to arbitrary types as long as they satisfy two conditions:

- The Paterson Conditions: type variables in the assertion cannot have more occurrences than in the head. Moreover, the assertion must have fewer constructors and variables than the head.
- The Coverage Condition: If we consider a substitution S mapping each type variable in the class declaration to the corresponding type in the instance declaration, every functional dependency `x1..xn -> y1..ym`, must be such that every type variable in S(`y1..yn`) appears in S(`x1..xn`).

These restrictions are needed to ensure that context reduction terminates.

UndecidableInstances: Lifts the aforementioned restrictions on instance heads, by not guaranteeing termination of context reduction, and thus type checking. For instance, it will allow us to write the instance:

```
instance C a => C a where
```

Reducing the context (`C a`) would then make the typechecker loop. This extension should be used with care, as it is up to the programmer to make sure that context reduction will terminate. In practice, GHC will impose a stack size limit to ensure termination of typechecking.

OverlappingInstances: GHC requires that instance resolution should be unambiguous, when solving a type-class constraint. Therefore, if there are two instances that match a given constraint, it will report an error. This can be relaxed by allowing instances to overlap, provided that there is a most specific one. In this case, GHC will always choose the most specific instance.

However, if there is any potential for the instance to change, through polymorphic application, GHC will not accept the instance.

IncoherentInstances: This extension allows us to specify that the most specific instance possible should always be selected, even if there is potential for that to change in further instantiations of the same term.

Examples of their uses are mentioned by Jones et al. in [JJM97].

2.2.1 Type Families

Functional dependencies allow us to specify functional dependencies in the parameters of a multi-parameter type class. However, it is commonly the case that what is actually meant by the functional dependencies is that we would like an associated type to the type class, i.e. a function from the type parameters of the type class to another type. For instance, in the `Collects` example:

```
class Collects e ce | ce -> e where
...
```

We are expressing the fact that for every collection `ce`, there is a unique type for the element `e`. Based on how common this pattern is, Schrijvers et al. [SPJCS08a] propose an alternative, as type synonyms associated with a class. They extend the type class notation with the possibility of stating associated type synonyms, with the same syntax as type synonyms. In our example, `Collects` would be expressed as:

```
class Collects ce where
  type Elem e
  ...
```

This notation arguably reflects what is meant more clearly. Every instance would then give an implementation for Elem, e.g.

```
instance Eq e => Collects [e] where
  type Elem [e] = e
  ...
```

`Elem` can then be thought as a type-level function, of kind $* \rightarrow *$.

2.2.2 Embedding Type Systems

When specifying function types that are less generic than their principal type, we are specifying constraints on what types it can be used with. For example, if we specify:

```
plusInt :: Int -> Int -> Int
plusInt = (+)
```

We are stating that this function can only be used to add values of type *Int*. When defining our own data types, it can be useful to allow information to be embedded on the type level. For instance, consider an embedded expression language:

```
data Expr = ValE Integer | AddE Expr Expr | StringE String
```

The type of the constructor `AddE` is arguably too generic, as it will allow us to apply `AddE` to strings. We can thus think of constraining it by defining a family of smart constructors, and hiding the original data constructors using Haskell's module system:

```
valE :: ?
valE = ValE

addE :: ?
addE = AddE

stringE :: ?
stringE = StringE
```

We want to constrain the type of `addE` so that only values that are effectively integers will be allowed. In order to do so at the type level, we need more information about e.g. `valE 3`, than just `Expr`. Thus, it can be useful to parametrise the data type with another type:

```
data Expr a = ValE Integer | AddE (Expr a) (Expr a) | StringE String
```

We can now define smart constructors so that we embed typing information about our language:

```
valE :: Expr Integer
valE = ValE

addE :: Expr Integer -> Expr Integer -> Expr Integer
addE = AddE

stringE :: Expr String
stringE = StringE
```

The term `(addE (stringE "a") (stringE "b"))` is now ill-typed. Note that the choice of *Integer* and *String* as the values for our type parameter is arbitrary (even if meaningful), as they are not linked to the type of any value. We are just using them as type-level constants. GHC allows us to define type level constants as empty data types, with the extension `EmptyDataDecls`. For instance, we could define:

```
data IntegerT
data StringT
```

In that case, and without losing any safety we could define:

```
valE :: Expr IntegerT
valE = ValE

addE :: Expr IntegerT -> Expr IntegerT -> Expr IntegerT
addE = AddE

stringE :: Expr StringT
stringE = StringE
```

This example shows how one can embed a type system by parameterising a data type and using that parameter to build constraints in the types of functions.

2.2.3 GADTs

While the previous example allowed us to constrain our data type, it seems redundant to define data constructors that are there just to introduce the data type, and then define smart constructors to restrict its usage. Generalized algebraic data types (GADTs) were thus introduced in GHC to allow us to constrain the types of data constructors arbitrarily. With GADTs, we can write the previous expression language as:

```
data Expr a where
  ValE :: Expr IntegerT
  AddE :: Expr IntegerT -> Expr IntegerT -> Expr IntegerT
  StringE :: Expr StringT
```

One advantage of this mechanism is that the constraints between the data constructors and the types they will be assigned are intrinsic to the constructors themselves. Thus, they can be brought into context by pattern matching. Consider for instance, length-indexed vectors:

```
data Zero
data Succ a

data Vec a n where
  VecNil  :: Vec a Zero
  VecCons :: a -> Vec a n -> Vec a (Succ n)
```

If one were to define a head function that would only operate on non-empty lists (i.e. $\forall n.\ Vec\ a\ (Succ\ n)$):

```
vecHead :: Vec a (Succ n) -> a
vecHead (VecCons x xs) = x
```

GHC would not report this as a non-exhaustive pattern match, as $VecNil$ has the constraint that $n = Zero$, while this function will only take vectors with $n = (Succ\ n')$. In fact adding a case for $VecNil$ would be a type error.

2.2.4 Promotion and Richer Kinds

In the previous section we introduced Peano number constants at the type level in order to parameterise a vector type by a natural number. What we actually meant to do was to introduce a data type at the type-level. In order to allow more natural definitions of this kind (somewhat mirroring dependently typed languages, but while still enforcing a value-type separation), Yorgey et al. [YWC+12] introduce promotion of data types and richer kinds (making the kind level rich enough to mirror the type level). Thus, when writing:

```
data Nat = Zero | Succ Nat
```

We are actually also defining a kind `Nat`, and the type-level `Zero` and `Succ` as before, with the exception that they have kind `Nat` instead of `*`. Whenever a polymorphic data constructor, such as `[] :: [a]`, is promoted, kind polymorphism is induced, i.e. we get `'[] :: [k]`. Here the quote is needed to distinguish the list type from a type-level list.

2.3 Discussion

In the beginning of this chapter we introduced the lambda calculus, as a calculus embodying the core concepts of functional programming, which is nevertheless powerful enough to express all computable functions. We then showed how we can apply type systems to it, to ensure that the programs that we write will not get stuck in execution. This calculus forms a solid metatheoretical base to experiment with the functional programming paradigm, and seeing how adding different features affects the properties that it exhibits.

We then introduced Haskell as a practical platform for performing this experimentation within certain well-specified boundaries. Haskell achieves this by providing the means to embed some alternative type systems, through the combination of generalized algebraic data types and the type class mechanism. These allow us to define logical relations between types, reflecting domain-specific logics. The Haskell type system is strong enough to ensure that even in the presence of these additions, the resulting system is still type sound.

CHAPTER 3

Related Work

Within the research literature there are three main approaches to the design of a development environment for developing context-aware applications:

- The most practical approach, stemming from the Ubiquitous computing field, is to create a library of common functionality for context-aware applications, such as sensing, communication, processing and aggregation. This is done in current practical programming languages such as Java, and usually provides very few compile-time guarantees, in some cases bypassing even the type system of the host language. Examples of this approach are the Context Toolkit [SDA99] and JCAF [Bar05].
- Another approach closely related to the first, is to attempt to use a general purpose programming language to embed this functionality, but where the language provides ample capabilities for type-level programming. This is what we will do with our Haskell library for context-dependent values.
- The more theoretical approach is to develop a programming language from the ground up with these capabilities in mind. This has the advantage that we are able to provide thorough compile-time guarantees. This is the approach we take with λ_{env}, a lambda-calculus with support for context-dependent types. Other approaches in this area include context-oriented programming [HCN08], which traditionally has been embodied in modifications of object-oriented calculi, such as FJ with ContextFJ [IHM12].

3.1 Pervasive Computing Frameworks

In the Ubiquitous computing research literature, there are already frameworks available for developing context-aware applications, with a variety of features. One of the most influential

frameworks is the Context Toolkit [DA00a]. This toolkit was significantly different from the approaches taken before it in that it decoupled the sensing units into widgets, providing access to this data through a networked API. It also has a notion of interpreting raw data into context. As such, it is a good example of how a context server architecture encourages the addition of interpretation to the context providers and can make context user applications more lightweight. It also incorporates discovery of contextual widgets.

From the point of view of our main thesis, that lightweight programming language constructs that preserve static typing information are essential for context-awareness to truly become widespread, the context toolkit represents the polar opposite. A context-aware application has to directly communicate with the context toolkit components and switch between the context toolkits' key/value pairs and its own explicitly. However, its main contribution is in the usage of a widget abstraction for sensing components and providing a ubiquitous networked communications interface. This provides a tremendous amount of flexibility in application development but is also rather unsafe. For instance, "type" information has to be explicitly dealt with at the value level, and there is no compiler support for checking that it is handled soundly. Thus, even though it is defined within a strongly typed object-oriented language, the amount of type checking that is undertaken by the compiler is minimal, with most of it happening at runtime. This is akin to switching to dynamic typing in order to encode the variability we require of the application. The system defined in this thesis is not going to reach the full set of features that the context toolkit provides, but it will mitigate some of its downfalls. The following code snippet comes from the context toolkit documentation [DA] and illustrates the usage of strings and value-level "types":

```
1  BaseObject server = new BaseObject(7777); // create BaseObject running
       on port 7777
2
3  Attributes subAtts = new Attributes();
4  subAtts.addAttribute(WPersonPresence.USERID);
5  subAtts.addAttribute(WPersonPresence.TIMESTAMP);
6  Conditions subConds = new Conditions();
7  subConds.addCondition
       (WPersonPresence.USERID,Storage.EQUAL,"16AC850600000044");
8  Error error = server.subscribeTo(this, 7777, "testApp", "localhost",
       5555, "PersonPresence_here",
9                  WPersonPresence.UPDATE, "presenceUpdate",
                       subConds,subAtts);
10 System.out.println("Subscription␣with␣valid␣attributes/conditions:␣
```

```
        "+error6.getError());
11
12    ...
13
14    public DataObject handle(String callback, DataObject data) throws
          InvalidMethodException, MethodException {
15      if (callback.equals("presenceUpdate")) {
16        AttributeNameValues atts = new AttributeNameValues(data);
17        AttributeNameValue timeAtt =
              atts.getAttributeNameValue(WPersonPresence.TIMESTAMP);
18        String time = (String)timeAtt.getValue();
19        System.out.print(time+"\n");
20      }
21    }
```

Listing 3.1: Context Toolkit code example.

Note how the application code has to be polluted by references to the communication infrastructure and the usage of a separate ad-hoc runtime "type" system. The application model is event-driven and callback based. The widget's user subscribes to a particular widget, defines what properties it is interested in and under what conditions it should be notified. Again from the point of view of typing, all contextual information is cast to a supertype, and has to be manually downcast. Moreover, there is no ontology definition and in fact strings are used to identify contextual values. This string comparison is particularly error prone and offers no compile-time checking.

In this thesis, I will attempt to bring some of this flexibility while staying within the constraints of the type system. More concretely, I will define a specification for the global information that is provided to the application, as well as an embedded type system for it, which is missing in the context toolkit. Then, I will attempt to add the flexibility that we get by casting to a supertype, by identifying some of the circumstances under which this would be useful and encoding them within a type system.

3.2 Context-Oriented Programming

As mentioned in the overview, on the other side of the spectrum, we have approaches that, similarly to ours, attempt to embody the logic of context-awareness into a programming language and thus provide type soundness. Context-oriented programming is one such move-

ment.

Context-oriented programming [HCN08] models context-awareness within the framework of object-oriented method dispatch by adding another dimension to the dispatch, that of context. In order to do so, it allows the programmer to define context-based method overrides called layers. Method dispatch is then dependent on the name of the message, sender, receiver and context, forming what they name *four-dimensional dispatch*. A context comprises a set of active layers. These layers are activated and deactivated explicitly through the use of **with** and **without** scoping constructs, respectively. Layer activation from environmental information is also taken into account in some implementations, by registering the layers that a particular environmental reading activates and for what methods. Another feature that is described but not present in all implementations is the ability to deactivate layers using a scoping **without** construct.

Similarly to the **super** keyword provided by subclass overriding, context-oriented programming adds a **proceed** keyword, allowing execution to proceed with the next active layer or the original implementation. Another feature that is commonly added is that of layers adding methods to classes. In this case, a notion of layer requirement needs to be defined, whereby methods are defined only on the condition that certain layers are active. The paper that outlines the requirements [HCN08] does not provide a formal model for this behaviour but it does provide several implementations in Java, Smalltalk and Common Lisp. Since the original paper there have been several attempts at formalising context-oriented programming, mostly by extending Featherweight Java, a minimal core calculus for Java [IPW01]. We take a small detour here to present the rules of Featherweight Java, so we can see how it is affected by the context-oriented programming extensions and to show how these perform in terms of our design goals.

3.2.1 Featherweight Java

Featherweight Java (FJ) [IPW01] is a fairly compact minimal core calculus for Java. Its syntax is a subset of Java:

$$
\begin{aligned}
&CL ::= \textbf{class}\ C\ \textbf{extends}\ C\{\overline{C\ f};\ K\ \overline{M}\}\\
&K ::= C(\overline{C\ f})\{super(\overline{f});\ \textbf{this}.f = f\}\\
&M ::= C\ m(\overline{C\ x})\{\ \textbf{return}\ t;\}\\
&t ::= x \mid t.f \mid t.m(\overline{t}) \mid \textbf{new}\ C(\overline{t}) \mid (C)t\\
&v ::= \textbf{new}\ C(\overline{v})
\end{aligned}
$$

FJ also defines a class table, as a mapping from class names C to class declarations CL ($C \rightarrow CL$). A program is then a pair (CT, t), of a class table and a term. FJ defines lookups over the class table for fields, method types and method bodies. In the method type/body lookup definition, we can see that dispatching is done on the type of the receiver, proceeding to the direct supertype of that type, if it is not an overridden method, until it is found. This is the main part that will have to be modified to change the dispatching technique.

Fields: $\boxed{fields(C) = \overline{C\ f}}$

$$\frac{}{fields(\textbf{Object}) = \bullet}\ \textsc{FldObj}$$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\{\overline{C\ f}; K\ \overline{M}\} \quad fields(D) = \overline{D\ g}}{fields(C) = \overline{D\ g}, \overline{C\ f}}\ \textsc{FldClass}$$

Method types: $\boxed{mtype(m, C) = T}$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\{\overline{C\ f}; K\ \overline{M}\} \quad B\ m(\overline{B\ x})\{\ \textbf{return}\ t;\} \in \overline{M}}{mtype(m, C) = \overline{B} \rightarrow B}\ \textsc{MTypeClass}$$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\{\overline{C\ f}; K\ \overline{M}\} \quad m\ \textbf{is not defined in}\ \overline{M}}{mtype(m, C) = mtype(m, D)}\ \textsc{MTypeSuper}$$

Method bodies: $\boxed{mbody(m, C) = (\overline{x}, t)}$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\{\overline{C\ f}; K\ \overline{M}\} \quad B\ m(\overline{B\ x})\{\ \textbf{return}\ t;\} \in \overline{M}}{mbody(m, C) = (\overline{x}, t)}\ \textsc{MBodyClass}$$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\{\overline{C\ f}; K\ \overline{M}\} \quad m\ \textbf{is not defined in}\ \overline{M}}{mbody(m, C) = mbody(m, D)}\ \textsc{MBodySuper}$$

An auxiliary subtyping relation is also defined based on the syntactical subclassing present in the class table, as the reflexive transitive closure of the explicitly declared subtyping relations:

$\boxed{C <: C'}$

$$\frac{}{C <: C}\ \textsc{SubtypRefl}$$

$$\frac{C <: D \qquad D <: E}{C <: E}\ \textsc{SubtypTrans}$$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\ \{...\}}{C <: D}\ \textsc{SubtypDecl}$$

Using these auxiliary definitions, evaluation is then defined as follows. Again, note that dispatching is delegated to the *mbody* auxiliary lookup, so these rules are fairly straightforward. In Featherweight Java, dispatching on the type of the receiver is encoded by having the object runtime value remain tagged with its runtime type, as a **new** term. Having kept this information it is then easy to dispatch on the runtime type of the receiver.

$\boxed{t \longrightarrow t'}$

$$\frac{fields(C) = \overline{C\ f}}{(\textbf{new}\ C(\overline{v})).f_i \longrightarrow v_i}\ \textsc{EvalFld}$$

$$\frac{mbody(m, C) = (\overline{x}, t)}{(\textbf{new}\ C(\overline{v})).m(\overline{u}) \longrightarrow [\overline{x} \mapsto \overline{u}, \textbf{this} \mapsto \textbf{new}\ C(\overline{v})]t}\ \textsc{EvalMBody}$$

$$\frac{C <: D}{(D)(\textbf{new}\ C(\overline{v})) \longrightarrow \textbf{new}\ C(\overline{v})}\ \textsc{EvalCast}$$

Typing in FJ is then fairly standard as well, having defined the lookups and auxiliary subtyping relation. Of interest here, regarding our design goals, is the fact that overriding is not allowed to change the return types. This implies that the overriding metaphor, as seen in object-oriented programming, is not entirely adequate to formulate context-awareness in the way we defined it in the introduction.

$\boxed{\Gamma \vdash t : C}$

$$\frac{x : C \in \Gamma}{\Gamma \vdash x : C}\ \textsc{TypVar}$$

$$\frac{\Gamma \vdash t : C \qquad fields(C) = \overline{C\ f}}{\Gamma \vdash t.f_i : C_i}\ \textsc{TypField}$$

$$\frac{\Gamma \vdash t : C \quad mtype(m, C) = \overline{D} \to C \quad \overline{\Gamma \vdash t_i : C_i}^{\,i} \quad \overline{C <: D}}{\Gamma \vdash t.m(\bar{t}) : C} \text{ TypInvk}$$

$$\frac{fields(C) = \overline{D\ f} \quad \overline{\Gamma \vdash t_i : C_i}^{\,i} \quad \overline{C <: D}}{\Gamma \vdash \textbf{new}\, C(\bar{t}) : C} \text{ TypNew}$$

$$\frac{\Gamma \vdash t : D \quad D <: C}{\Gamma \vdash (C)t : C} \text{ TypUCast}$$

$$\frac{\Gamma \vdash t : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)t : C} \text{ TypDCast}$$

$$\frac{\Gamma \vdash t : D \quad C \not<: D \quad D \not<: C \quad \textbf{stupid warning}}{\Gamma \vdash (C)t : C} \text{ TypSCast}$$

All of these rules are predicated on well-formedness constraints on classes, that ensure that the class definitions are sound with regards to subtyping and declared types matching the types of the terms.

$\boxed{CL\ \textbf{OK}}$

$$\frac{K = C(\overline{D\ g}, \overline{C\ f})\{super(\overline{g}); \overline{\textbf{this}.f = f}\} \quad fields(D) = \overline{D\ g} \quad \overline{M\ \textbf{OK IN}\ C}}{\textbf{class}\, C\, \textbf{extends}\, D\{\overline{C\ f}; K\ \overline{M}\}\ \textbf{OK}} \text{ WFCL}$$

$\boxed{M\ \textbf{OK IN}\ C}$

$$\frac{\begin{array}{l}\overline{x : C}, \textbf{this} : C \vdash t : E \\ E <: C' \\ CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\ \{...\} \\ override(m, D, \overline{C} \to C')\end{array}}{C'\ m(\overline{C\ x})\{\ \textbf{return}\ t;\}\ \textbf{OK IN}\ C}\ \text{WFM}$$

$\boxed{override(m, D, \overline{C} \to C)}$

$$\frac{mtype(m, D) = \overline{D} \to D\ \textbf{implies}\ \overline{C} = \overline{D}\ \textbf{and}\ C = D}{override(m, D, \overline{C} \to C)}\ \textsc{Override}$$

3.2.2 `with/without` ContextFJ

Dave Clarke and Ilya Sergey [CS09] present the first formal dynamic semantics for context-oriented programming. Their calculus faithfully represents the requirements that were set out in the context-oriented programming paper, and uses **with** / **without** keywords for explicit layer activation. Their calculus also allows for methods to be introduced by layers, and possesses a notion of method assumptions for this purpose. However, they only argue its type soundness, and present no proofs. Their type system turns out to be unsound as it does not handle removal of layers properly [IHM12].

Hirschfeld, Igarashi and Masuhara [HIM11] present a type-sound context-oriented programming extension for FJ. They preserve class-based overriding of methods, but remove the possibility for layers to introduce new methods. According to the authors, this is the first direct semantics of context-oriented programming features. Their grammar is the following:

$CL ::= \textbf{class}\ C\ \textbf{extends}\ C\{\overline{C\ f}; K\ \overline{M}\}$

$K ::= C(\overline{C\ f})\{super(\overline{f}); \overline{\textbf{this}.f\ =\ f}\}$

$M ::= C\ m(\overline{C\ x})\{\ \textbf{return}\ t;\}$

$t ::= x \mid t.f \mid t.m(\bar{t}) \mid \textbf{new}\ C(\bar{t}) \mid \textbf{with}\ L\ t \mid \textbf{without}\ L\ t \mid \textbf{proceed}\ (\bar{t}) \mid \textbf{super}\ .m(\bar{t})$
$\quad \mid \textbf{new}\ C(\overline{v}) < C, \overline{L}, \overline{L} > .m(\bar{t})$

$v ::= \textbf{new}\ C(\overline{v})$

They store partial methods in a partial method table PT, mapping lookup triples of methods, class and layer names to the body of the method ($m \times C \times L \to M$). They remove casts from FJ, and add the **with** / **without** / **proceed** keywords specified in the original COP paper. They also extend FJ with a **super** keyword for method invocation on the superclass within arbitrary expressions. In addition to this, to keep track of the accummulated labels within a

direct semantics, they introduce a runtime value: **new** $C(\overline{v}) < D, \overline{L'}, \overline{L} > .m(\overline{t})$. representing a method call with additional information for method lookup. $\overline{L'}$ is assumed to be a prefix of $\overline{L}$. Essentially, the lookup will proceed within class D, through the labels in $\overline{L'}$ until the point where these are exhausted. It will then proceed with the direct superclass of D, reinitialising $\overline{L'}$ to $\overline{L}$. This is needed to give a semantics to **super** and **proceed**. This is reflected in the modified rules for method lookup:

$\boxed{mbody(m, C, \overline{L}', \overline{L}) = (\overline{x}, t)}$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\{\overline{C\ f}; K\ \overline{M}\} \quad B\ m(\overline{B\ x})\{\ \textbf{return}\ t;\} \in \overline{M}}{mbody(m, C, \bullet, \overline{L}) = \overline{x}.t\ \textbf{in}\ C, \bullet} \quad \textsc{MBodyClass}$$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\{\overline{C\ f}; K\ \overline{M}\} \quad m\ \textbf{is not defined in}\ \overline{M}}{mbody(m, C, \bullet, \overline{L}) = mbody(m, D, \overline{L}, \overline{L})} \quad \textsc{MBodySuper}$$

$$\frac{PT(m, C, L) = C\ m(\overline{C\ x})\{\ \textbf{return}\ t;\}}{mbody(m, C, \overline{L}'; L, \overline{L}) = \overline{x}.t\ \textbf{in}\ C, \overline{L}'; L} \quad \textsc{MBodyLayer}$$

$$\frac{PT(m, C, L)\ \textbf{undefined}}{mbody(m, C, \overline{L}'; L, \overline{L}) = mbody(m, C, \overline{L}', \overline{L})} \quad \textsc{MBodyNextLayer}$$

In order to keep track of the activated layers in a direct style, they change the operational semantics judgements to include a layer environment. The judgements are of the form $\overline{L} \vdash t \longrightarrow t'$ and mean that term t reduces to term t' under the activated layers $\overline{L}$. $\overline{L}$ must not contain duplicated names. This is ensured by removing a layer before it is re-added to the list.

$\boxed{\overline{L} \vdash t \longrightarrow t'}$

$$\frac{fields(C) = \overline{C\ f}}{\overline{L} \vdash (\textbf{new}\ C(\overline{v})).f_i \longrightarrow v_i} \quad \textsc{Fld}$$

$$\frac{\overline{L} \vdash \textbf{new}\ C(\overline{v}) < C, \overline{L}, \overline{L} > .m(\overline{v}) \longrightarrow t}{\overline{L} \vdash \textbf{new}\ C(\overline{v}).m(\overline{v}) \longrightarrow t} \quad \textsc{MthInvkInit}$$

$$\frac{\begin{array}{l} mbody(m, C', \overline{L}', \overline{L}) = \overline{x}.t \textbf{ in } C'', \bullet \\ CT(C'') = \textbf{class } C'' \textbf{ extends } D \{...\} \end{array}}{\overline{L}''' \vdash \textbf{new } C(\overline{v}) < C', \overline{L}', \overline{L} > .m(\overline{v}) \longrightarrow \left[\begin{array}{l} \overline{x} \mapsto \overline{v}, \\ \textbf{this} \mapsto \textbf{new } C(\overline{v}), \\ \textbf{super} \mapsto \textbf{new } C(\overline{v}) < D, \overline{L}, \overline{L} > \end{array}\right] t} \text{ MthInvk}$$

$$\frac{\begin{array}{l} mbody(m, C', \overline{L}', \overline{L}) = \overline{x}.t \textbf{ in } C'', \overline{L}''; L \\ CT(C'') = \textbf{class } C'' \textbf{ extends } D \{...\} \end{array}}{\overline{L}''' \vdash \textbf{new } C(\overline{v}) < C', \overline{L}', \overline{L} > .m(\overline{v}) \longrightarrow \left[\begin{array}{l} \overline{x} \mapsto \overline{v}, \\ \textbf{this} \mapsto \textbf{new } C(\overline{v}), \\ \textbf{super} \mapsto \textbf{new } C(\overline{v}) < D, \overline{L}, \overline{L} >, \\ \textbf{proceed} \mapsto \textbf{new } C(\overline{v}) < C'', \overline{L}'', \overline{L} > .m \end{array}\right] t''} \text{ MthInvk2}$$

$$\frac{\begin{array}{l} \textbf{remove } (L, \overline{L}) = \overline{L}' \\ \overline{L}'; L \vdash t \longrightarrow t' \end{array}}{\overline{L} \vdash \textbf{with } L\, t \longrightarrow \textbf{with } L\, t'} \text{ With}$$

$$\frac{\begin{array}{l} \textbf{remove } (L, \overline{L}) = \overline{L}' \\ \overline{L}' \vdash t \longrightarrow t' \end{array}}{\overline{L} \vdash \textbf{without } L\, t \longrightarrow \textbf{without } L\, t'} \text{ Without}$$

$$\frac{}{\overline{L} \vdash \textbf{with } L\, v \longrightarrow v} \text{ WithVal}$$

$$\frac{}{\overline{L} \vdash \textbf{without } L\, v \longrightarrow v} \text{ WithoutVal}$$

The type system judgements are modified as well, to keep track of the structure under which a certain expression appears. It can either be $\bullet$ which stands for execution, $C.m$, which means it appears in method m of class C, or $L.C.m$, meaning m in class C in layer L. It affects the typing rules for **super** and **proceed** , by providing information about what class/layer execution will be delegated to. It also makes sure that these constructs can only appear where they are indeed defined.

$\boxed{\mathscr{L}; \Gamma \vdash t : C}$

$$\frac{x : C \in \Gamma}{\mathscr{L}; \Gamma \vdash x : C} \text{ TypVar}$$

$$\frac{\begin{array}{l} \mathscr{L}; \Gamma \vdash t : C \\ fields(C) = \overline{C\ f} \end{array}}{\mathscr{L}; \Gamma \vdash t.f_i : C_i} \text{ TypField}$$

$$\frac{\begin{array}{l}\mathscr{L};\Gamma \vdash t : C \\ mtype(m, C) = \overline{D} \to D \\ \overline{\mathscr{L};\Gamma \vdash t_i : E_i}^{\,i} \\ \overline{E <: D}\end{array}}{\mathscr{L};\Gamma \vdash t.m(\bar{t}) : D}\ \textsc{TypInvk}$$

$$\frac{\begin{array}{l}fields(C) = \overline{D\ f} \\ \overline{\mathscr{L};\Gamma \vdash t_i : C_i}^{\,i} \\ \overline{C <: D}\end{array}}{\mathscr{L};\Gamma \vdash \mathbf{new}\, C(\bar{t}) : C}\ \textsc{TypNew}$$

$$\frac{\mathscr{L};\Gamma \vdash t : C}{\mathscr{L};\Gamma \vdash \mathbf{with}\, L\, t : C}\ \textsc{TypWith}$$

$$\frac{\mathscr{L};\Gamma \vdash t : C}{\mathscr{L};\Gamma \vdash \mathbf{without}\, L\, t : C}\ \textsc{TypWithout}$$

$$\frac{\begin{array}{l}\mathscr{L} = C.m\ \mathbf{or}\ L.C.m \\ CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ E\ \{...\} \\ mtype(m', E) = \overline{D} \to D \\ \overline{\mathscr{L};\Gamma \vdash t_i : E_i}^{\,i} \\ \overline{E <: D}\end{array}}{\mathscr{L};\Gamma \vdash \mathbf{super}.m'(\bar{t}) : D}\ \textsc{TypSuper}$$

$$\frac{\begin{array}{l}\mathscr{L} = L.C.m \\ mtype(m, C) = \overline{D} \to D \\ \overline{\mathscr{L};\Gamma \vdash t_i : E_i}^{\,i} \\ \overline{E <: D}\end{array}}{\mathscr{L};\Gamma \vdash \mathbf{proceed}\,(\bar{t}) : D}\ \textsc{TypProceed}$$

$$\frac{\begin{array}{l}fields(C) = \overline{D\ f} \\ \overline{\mathscr{L};\Gamma \vdash v_i : C_i}^{\,i} \\ \overline{C <: D} \\ C <: D \\ mtype(m, D) = \overline{F} \to F \\ \overline{\mathscr{L};\Gamma \vdash t_i : F_i}^{\,i} \\ \overline{E <: F}\end{array}}{\mathscr{L};\Gamma \vdash \mathbf{new}\, C(\overline{v}) < D, \overline{L}, \overline{L''} > .m(\bar{t}) : F}\ \textsc{TypNewRT}$$

Soundness is again predicated on the well-formedness of the class table, as in FJ, except for

the addition of the $\mathscr{L}$ context. In particular, regarding our design goals, it should be noted that layers are not allowed to change the return types of overriden methods (modulo subtyping). This is faithful to the object-oriented overriding metaphor.

3.2.3 `ensure` ContextFJ

In Igarashi, Hirschfeld and Masuhara's follow-up work [IHM12], they present an alternative calculus for context-oriented programming, whereby they replace the **with** keyword with an **ensure** keyword. The difference between these two is that activating an already active layer with **with** would change the dispatch order for the expression by placing the newly activated layer as the first one in the dispatch order. **ensure** on the other hand, will have no effect on a layer that is already active. The motivation for this is that changing the order of the activated layers may break requirements between the layers. For instance, if we have the configuration $L_1; L_2; L_3$ where L_3 requires L_1, and we activate L_1, L_3's requirement will no longer be satisfied. Since their type-system uses a static approximation of the layers that will be active, this would be unsound. Likewise, they remove the **without** keyword, as this could lead to a similar problem. With these restrictions, they allow layers to add methods to classes, which their previous system disallowed. Using this technique they prove type soundness of their system.

$\boxed{\overline{L} \vdash t \longrightarrow t'}$

$$\frac{fields(C) = \overline{C\ f}}{\overline{L} \vdash (\mathbf{new}\, C(\overline{v})).f_i \longrightarrow v_i}\ \textsc{Fld}$$

$$\frac{\overline{L} \vdash \mathbf{new}\, C(\overline{v}) < C, \overline{L}, \overline{L} > .m(\overline{v}) \longrightarrow t}{\overline{L} \vdash \mathbf{new}\, C(\overline{v}).m(\overline{v}) \longrightarrow t}\ \textsc{MthInvkInit}$$

$$\frac{\begin{array}{c} mbody(m, C', \overline{L}', \overline{L}) = \overline{x}.t\ \mathbf{in}\ C'', \bullet \\ CT(C'') = \mathbf{class}\ C''\ \mathbf{extends}\ D\ \{...\} \end{array}}{\overline{L}''' \vdash \mathbf{new}\, C(\overline{v}) < C', \overline{L}', \overline{L} > .m(\overline{v}) \longrightarrow \left[\begin{array}{l} \overline{x} \mapsto \overline{v}, \\ \mathbf{this} \mapsto \mathbf{new}\, C(\overline{v}), \\ \mathbf{super} \mapsto \mathbf{new}\, C(\overline{v}) < D, \overline{L}, \overline{L} > \end{array}\right] t}\ \textsc{MthInvk}$$

$$\frac{\begin{array}{l} mbody(m, C', \overline{L}', \overline{L}) = \overline{x}.t \textbf{ in } C'', \overline{L}''; L \\ CT(C'') = \textbf{class } C'' \textbf{ extends } D \{...\} \end{array}}{\overline{L}''' \vdash \textbf{new } C(\overline{v}) < C', \overline{L}', \overline{L} > .m(\overline{v}) \longrightarrow \left[\begin{array}{l} \overline{x} \mapsto \overline{v}, \\ \textbf{this} \mapsto \textbf{new } C(\overline{v}), \\ \textbf{super} \mapsto \textbf{new } C(\overline{v}) < D, \overline{L}, \overline{L} >, \\ \textbf{proceed} \mapsto \textbf{new } C(\overline{v}) < C'', \overline{L}'', \overline{L} > .m \end{array}\right] t''} \text{ MthInvk2}$$

$$\frac{\begin{array}{c} \textbf{ensure } (L, \overline{L}) = \overline{L}' \\ \overline{L}'; L \vdash t \longrightarrow t' \end{array}}{\overline{L} \vdash \textbf{ensure } L\, t \longrightarrow \textbf{ensure } L\, t'} \text{ Ensure}$$

$$\frac{}{\overline{L} \vdash \textbf{ensure } L\, v \longrightarrow v} \text{ EnsureVal}$$

$$\textbf{ensure } (L, \overline{L}) = \overline{L}, \text{if } L \in \overline{L}$$
$$\textbf{ensure } (L, \overline{L}) = \overline{L}; L, \text{otherwise}$$

The type system judgements are again modified to add a set of layers Λ, which represents an under-approximation of the set of layers that will be active at a certain point in the program. This is used to make sure that the required layers for a given method are really activated when the method is called.

$\boxed{\mathscr{L}; \Lambda; \Gamma \vdash t : C}$

$$\frac{x : C \in \Gamma}{\mathscr{L}; \Lambda; \Gamma \vdash x : C} \text{ TypVar}$$

$$\frac{\begin{array}{c} \mathscr{L}; \Lambda; \Gamma \vdash t : C \\ fields(C) = \overline{C\ f} \end{array}}{\mathscr{L}; \Lambda; \Gamma \vdash t.f_i : C_i} \text{ TypField}$$

$$\frac{\begin{array}{l} \mathscr{L}; \Lambda; \Gamma \vdash t : C \\ mtype(m, C, \Lambda) = \overline{D} \to D \\ \overline{\mathscr{L}; \Lambda; \Gamma \vdash t_i : E_i}^{\,i} \\ \overline{E <: D} \end{array}}{\mathscr{L}; \Lambda; \Gamma \vdash t.m(\bar{t}) : D} \text{ TypInvk}$$

$$\frac{\begin{array}{l} \mathit{fields}(C) = \overline{D\ f} \\ \overline{\mathscr{L}; \Lambda; \Gamma \vdash t_i : C_i}^{\,i} \\ \overline{C <: D} \end{array}}{\mathscr{L}; \Lambda; \Gamma \vdash \mathbf{new}\, C(\bar{t}) : C} \quad \textsc{TypNew}$$

$$\frac{\begin{array}{l} L\ \mathbf{req}\ \Lambda' \\ \Lambda' \subseteq \Lambda \\ \mathscr{L}; \Lambda \cup \{L\}; \Gamma \vdash t : C \end{array}}{\mathscr{L}; \Lambda; \Gamma \vdash \mathbf{ensure}\, L\, t : C} \quad \textsc{TypEnsure}$$

$$\frac{\begin{array}{l} \mathscr{L} = C.m \\ CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ E\ \{...\} \\ \mathit{mtype}(m', E, \varnothing) = \overline{D} \to D \\ \overline{\mathscr{L}; \Lambda; \Gamma \vdash t_i : E_i}^{\,i} \\ \overline{E <: D} \end{array}}{\mathscr{L}; \Lambda; \Gamma \vdash \mathbf{super}.m'(\bar{t}) : D} \quad \textsc{TypSuperB}$$

$$\frac{\begin{array}{l} \mathscr{L} = L.C.m \\ CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ E\ \{...\} \\ L\ \mathbf{req}\ \Lambda' \\ \mathit{mtype}(m', E, \Lambda' \cup \{L\}) = \overline{D} \to D \\ \overline{\mathscr{L}; \Lambda; \Gamma \vdash t_i : E_i}^{\,i} \\ \overline{E <: D} \end{array}}{\mathscr{L}; \Lambda; \Gamma \vdash \mathbf{super}.m'(\bar{t}) : D} \quad \textsc{TypSuperP}$$

$$\frac{\begin{array}{l} \mathscr{L} = L.C.m \\ L\ \mathbf{req}\ \Lambda' \\ \mathit{mtype}(m, C, \Lambda', \Lambda' \cup \{L\}) = \overline{D} \to D \\ \overline{\mathscr{L}; \Lambda; \Gamma \vdash t_i : E_i}^{\,i} \\ \overline{E <: D} \end{array}}{\mathscr{L}; \Lambda; \Gamma \vdash \mathbf{proceed}\,(\bar{t}) : D} \quad \textsc{TypProceed}$$

$$\frac{\begin{array}{l}\overline{L}' \textbf{ is a prefix of } \overline{L}'' \\ \overline{L}'' \textbf{ wf} \\ fields(C) = \overline{D\ f} \\ \overline{\mathscr{L}; \Lambda; \Gamma \vdash v_i : C_i}^{\,i} \\ \overline{C <: D} \\ C <: D \\ mtype(m, D, \{\overline{L}'\}, \{\overline{L}''\}) = \overline{F} \to F \\ \overline{\mathscr{L}; \Lambda; \Gamma \vdash t_i : F_i}^{\,i} \\ \overline{E <: F}\end{array}}{\mathscr{L}; \Lambda; \Gamma \vdash \textbf{new}\, C(\overline{v}) < D, \overline{L}, \overline{L}'' > .m(\overline{t}) : F} \quad \textsc{TypNewRT}$$

$\boxed{mtype(m, C, \Lambda, \Lambda) = T}$

$$\frac{\begin{array}{l}CT(C) = \textbf{ class } C \textbf{ extends } D\{\overline{C\ f}; K\ \overline{M}\} \\ B\ m(\overline{B\ x})\{\textbf{ return } t;\} \in \overline{M}\end{array}}{mtype(m, C, \Lambda_1, \Lambda_2) = \overline{B} \to B} \quad \textsc{MTypeClass}$$

$$\frac{\begin{array}{l}L \in \Lambda_1 \\ PT(m, C, L) = C'\ m'(\overline{C\ x})\{\textbf{ return } t;\}\end{array}}{mtype(m, C, \Lambda_1, \Lambda_2) = \overline{C} \to C'} \quad \textsc{MTypePMethod}$$

$$\frac{\begin{array}{l}CT(C) = \textbf{ class } C \textbf{ extends } D\{\overline{C\ f}; K\ \overline{M}\} \\ m \textbf{ is not defined in } \overline{M} \\ \forall L \in \Lambda_1 . PT(m, C, L) \textbf{ undefined}\end{array}}{mtype(m, C, \Lambda_1, \Lambda_2) = mtype(m, D, \Lambda_2, \Lambda_2)} \quad \textsc{MTypeSuper}$$

The judgement L **req** Λ embodies layer requirement (when layers use methods introduced by other layers), and holds when the layers in Λ satisfy the requirements in L. A sequence of layers is well-formed, $\overline{L}$ **wf** if all the layer requirements for all the layers in the sequence are satisfied. It is worth noting how allowing layers to add methods to classes, as would be expected in an object-oriented programming overriding metaphor, makes it difficult to remain type sound in the presence of core concepts of COP, such as layer deactivation. Moreover, it should be noted this extended calculus also does not allow return types to be changed by an overriding layer.

3.2.4 Discussion

The programming model suggested by context-oriented programming is very appealing to developers, as it provides a great amount of flexibility in developing and abstracting context-dependent behaviour. Moreover, the layer overriding metaphor, along with a `proceed` keyword closely mirrors the subclass overriding and `super` keyword common in object-oriented programming languages. However, while the receiver of a `super` call can be statically determined (it is the direct superclass of the current class), the next layer in a `proceed` call is dependent on the layers that are active at the call site of a given method. In a similar vein to aspect-oriented programming, this flexibility comes at a cost, namely that the program's behaviour is spread throughout the code and in order to reason about a portion of code one has to consider a fairly complicated dispatch system. Moreover, all definitions are open for extension, and can only truly be resolved at runtime. This is highlighted by the fact that all attempts at formalising context-oriented programming into a type-sound programming language have struggled with this variability. More specifically, dynamic removal of layers in the presence of layers introducing new methods is still an unsolved problem.

Moreover, as noted throughout our presentation, the way that overriding works in object-oriented languages requires return types to not change on overriden methods. This behaviour is essential to preserve the semantic "is-a" metaphor that characterises subclassing. It also allows for the types to remain concise and manageable in the presence of overriding. It does however also imply that this abstraction is not entirely adequate to implement context-awareness in the way we described in the introduction.

In this thesis we propose the opposite approach, whereby contextual variations are local, and the type system indicates where they exist. Smarter types and type inference come together to make it much more manageable to understand programs. This still allows us to abstract away context-dependent behaviour whilst preserving a strong notion of encapsulation and abstraction. Moreover, we also focus on formalising the somewhat neglected environmental awareness aspect of context-oriented programming, whereby we can branch/dispatch depending on globally available information. We also allow the layers to have different argument and return types, keeping with our design goals. This would be severely complicated by the addition of COP behaviour, as the layer order that a given expression will be evaluated with is only known at runtime. Thus, the work we present in this thesis is not attempting to mirror context-oriented programming, but instead to suggest a different approach to embedding context-awareness in programming languages, in the vein of pure functional programming. In doing so, we introduce interesting new features such as context-dependent types and type inference which are not trivially achievable in COP approaches.

3.3 Implicit Arguments

There has also been prior research in modelling implicit arguments in a functional programming language, most notably that of Lewis et al [LLMS00], which is implemented in the Haskell compiler GHC as a compiler-specific extension to the standard. The Haskell EDSL approach we will present in chapter 5 shares some characteristics with this calculus, such as the implicit "floating out" of implicit arguments in composite computations. Thus, implicit arguments are conceptually strong enough to model the notion of context-dependent values. However, they do not allow us to externally specify an ontology by which all contextual values must abide, as we stated in the design goals. Thus, we do not use GHC's implicit arguments but instead embed our own. Our approach was designed from the ground up to be customised to typical use cases in context-awareness, and that is reflected in our choice of providing an external ontology, defining the types of context labels. This also allows us to make queries to the knowledge base more automatic, as only the label is required to project the necessary contextual information. In Lewis et al's solution [LLMS00] all implicit arguments have a name that identifies them, and it is up to the programmer to manage the assignment of values to names and scoping of those names.

Moreover, implicit arguments are not strong enough to model the notion of context-dependent types. One can use a tagged union type to simulate it, but the dependency between the contextual condition and the respective side of the union that is chosen is lost. For example, if we have (in Haskell) a term of type *Either* $(a \rightarrow c)$ $(b \rightarrow c)$ and a term of type *Either a b*, we have lost the information that says that if they are to be evaluated under the same context, both terms will be in the same branch, so we should be able to apply one to the other. We have thus lost composability. For a more comprehensive description of this mechanism and its shortcomings, see section 7.3.

3.4 Dependent Types

Another possible approach to solving our problem is to embed the desired features within a very expressive type system, such as a dependently typed lambda calculus. For instance, in Agda [Nor07], we could represent contextual dependencies as implicit dependent arguments. This would allow us to satisfy the design goal we described of having types depend on contexts. For instance we could have:

$$prog : \{tmp : Temperature\} \rightarrow \textbf{if } tmp > 20 \textbf{ then } T_1 \textbf{ else } T_2$$

	Light	*Comp*	*Ont*	*CDT*	*Safe*	*Inf*
Context Toolkit/JCAF	✗	✓	✓	✗	✗	✗
Context-Oriented Programming	✓	✓	✓	✗	✗	✗
FJ Context-Oriented Programming	✓	✓	✗	✗	✓	✗
Implicit Arguments	✓	✓	✗	✗	✓	✓
Dependent Types	✓	✓	✗	✓	✓	✗

Table 3.1: Analysis of related work with regards to the design goals.

Subject to defining a method for providing contextual implicit values, it would theoretically be possible to define a language that satisfies most of our goals. However, this would still not be trivial, and we would still be constrained by the fact that we are embedding the language in abstractions that do not exactly represent the behaviour that we want to encode. For instance, implicit arguments are once again identified by name, and it would be hard to define an ontology of types for context identifiers, without resorting to code generation and metaprogramming. Moreover, complete type inference is undecidable for unrestricted dependently typed systems [Bar92]. Type checking is also undecidable in general, and can only be made decidable by enforcing totality in the language, for example by restricting (co)recursion to be (co)primitive [Tur04]. This is because otherwise arbitrary expressions are allowed in the types, and have to be reduced for type checking. Thus, the work we present in Chapter 6 presents a domain-specific restriction of dependent types whereby we retain decidable type checking and inference without restricting the value level.

3.5 Discussion

The related work survey we presented shows that no current approach addresses the full set of design goals we set out to achieve. Table 3.1 compares the various approaches according to the set of design goals we established in the introduction. They were: lightweight context usage (*Light*), composability (*Comp*), separate reusable definition of ontology/typing/context runtime (*Ont*), context-dependent types (*CDT*), strong typing and safety (*Safe*) and type inference and equality/equivalence for context-equivalent types (*Inf*).

Such a comparison is always bound to be contentious as it depends on the interpretation of the design goals, so we will justify our choices below. Moreover, the choice of design goals is also subjective, and obviously was not the choice that was made in the design of the solutions we analysed in this chapter. Nevertheless, this is our attempt at having a more objective and concrete set of evaluation metrics.

Solutions stemming from the ubiquitous computing field, such as the Context Toolkit and

JCAF do allow us to define an ontology/context runtime specifying how context is to be structured, as they were designed exactly for the design of context-aware applications. However, as we mentioned in the detailed analysis, they represent the polar opposite of our goals, being extremely flexible, but not having any emphasis on type safety.

As far as context-oriented programming, we represent both the concept proposed in the original presentation [HCN08] as well as the subsequent formalisations of it as extensions to Featherweight Java [HIM11, IHM12]. In particular, it is worth noticing that the environmental dependency aspect of COP was neglected and left out of the formalisations. Moreover, it does not address the context-dependent types problem. However, it preserves type soundness and composability.

Dependent types could theoretically be used to develop a context-aware application featuring context-dependent types. However, context provision would have to be done manually as to the best of our knowledge no current approaches attempt to provide this functionality in a dependently typed setting. Moreover, type inference is undecidable, which implies that the developer would have to explicitly write types for every term, making practical usage of these techniques difficult.

CHAPTER 4

Functional Abstractions for Context-Awareness

One of our goals in this thesis is to present an interpretation of context-aware programming in a functional programming setting. In order to do so, in this Chapter we will examine one particular definition of context-awareness which highlights the difficulties present. We will then formalize this model taking as basis the lambda calculus we introduced in Chapter 2, and present a prototype for how we believe these concepts can be incorporated in functional languages.

Thus, in this Chapter we present:

- An informal argument whereby we relate concepts from functional programming to a conceptual model of context-aware computation. We will analyse how this model foresees certain difficulties in preserving abstraction and modularity in the presence of context-awareness. We will then propose a solution to mitigate this problem.
- A prototypical type system that formalises the concepts we present in the informal argument and will serve as a skeleton for defining other systems.
- An embedding of these concepts into a simply typed lambda calculus, which is suitable for defining libraries in current functional programming languages.
- An extended lambda calculus with direct semantics that embodies the features present in the prototypical type system, and which can be useful to analyse the metatheoretical implications of context-awareness in the properties of a functional programming language.

4.1 Motivation

Lieberman and Selker argued in [LS00] that context-aware application development would mean moving from the black-box model of computation, where we think of programs as black boxes, taking an input to an output, to a model where we allow implicit information known as context, to arbitrarily affect the computation:

Figure 4.1: A model of context-awareness.

In this situation, they argue that the black-box model would be invalidated, as applications need to be allowed to change their behaviours arbitrarily according to the context. They present the situation as a dichotomy between abstraction and context-awareness that always implies some trade-off. As we saw in the related work chapter, attempts at formalising unrestricted context-aware systems have struggled with type soundness or inference. However, we believe that it is possible to introduce some context-awareness into a formal model of abstraction, without compromising the principle of abstraction completely.

4.2 From Functions to Context-Aware Computations

Arguably the most defining feature of a functional programming language is support for first-class functions. That is, functions are not special objects that only the compiler knows about, but are actually pieces of data that can be passed around and manipulated. In his classical 1990 paper, John Hughes [Hug89] promotes first-class functions as the ability to effortlessly glue programs together without caring about their implementation details. This is usually encapsulated in a composition operator $\circ$, such that if we represent programs as functions f, g, we can compose those programs into a composite program, $g \circ f$, such that $(g \circ f)(x) = g(f(x))$. This means that we feed the output of program f into program g:

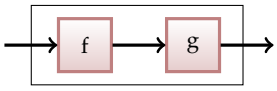

Figure 4.2: Regular function composition.

For example, if the input was an integer, the first box could be a computation that turns it into a list of all the numbers from 1 to that number, and the second box could be a computation

that multiplies all numbers in that list. The resulting program then calculates the factorial of the input value. Moreover, in typed functional programming languages, the compiler checks whether it makes sense to glue programs together. What this means is that each function will have a type: for instance $f : A \to B$ and $g : B \to C$ and the compiler knows that it only makes sense to compose two programs f and g together as $g \circ f$ if the codomain of f matches the domain of g. In our previous factorial example, if the first box (f), produces a list as its output (codomain), the second box (g) will have to accept lists as input (domain). Moreover, because we said that functions are first-class objects the operator $\circ$ does not need to be a custom operator, as it can just be seen as a function that takes two functions and produces a composite function. The composition operator $\circ$ will then, by definition, have the type: $((B \to C) \times (A \to B)) \to (A \to C)$. This allows us to treat functions as black boxes, and be able to compose them in complex ways, as if we were building a circuit diagram. In the previously described black box model of computation, the description of programs resembles our description of a function: a component that takes an input value and produces an output value. This is indeed the approach that is taken by most functional programming languages (with some extra complexity to handle side-effects, like state or I/O). Moreover, a value can be seen as a nullary function, that takes no input (or takes some input but ignores it) and always returns the same value:

Figure 4.3: A dataflow representation for constants.

With this description of programs, along with certain primitives for dealing with data, we can see how we can build programs purely by composition. We can also use the lambda calculus [Bar98] for defining the internals of our black boxes. Function application can be translated into composition easily (consider the constant example above). We can then still treat functions as black boxes after definition, using the composition operator, to build composite functions. This faithfully follows the principle of abstraction with regards to modularity and composability.

In our opinion, most of the currently existing context-awareness frameworks attempt to force this contextual information to become an input in the computation by sidestepping common abstraction conventions. The result is, as Lieberman and Selker pointed out, that modularity suffers. It is hard to use programs developed in context-aware frameworks without incorporating the entire framework in the application. The abstractions in the underlying languages are no longer sufficient to accommodate the flexibility required. We think that a more foundational approach could be taken, by developing a language that incorporates these ideas at its core.

Indeed, if we naïvely analyse the diagram that we presented for context-awareness, we will see that a context-aware computation is very similar to a regular function, but we have something that looks like an input, that we say has to be implicit. What it means for it to be implicit is that it does not need to be provided in order for the resulting value to be manipulated. It only needs to be resolved when we want to evaluate it. In this way we have something that looks like a function, but is not one, and composability appears to break as we apparently can not produce a black box that encapsulates both those functions. We believe that this does not have to be the case. We will start with a simple example, when the program in question takes no actual non-contextual input (or just ignores it). According to our description so far, this would be a program resembling:

Figure 4.4: Constants seen through the previous model.

An example of such a program would be a program that merely retrieves the user's location. This looks very similar to a function, however, as we said before the contextual input should be handled differently. We will see that there are operations similar to composition of context-aware computations that are meaningful, but they are not going to be the same. We argue for instance that in order to compose two computations that depend on context, the contextual dependency should be propagated outwards, as a dependency of the resulting black box:

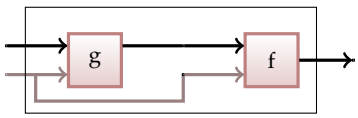

Figure 4.5: Composition of context-aware computations.

In this sense, composability is still valid. We can still combine context-aware computations easily, and the necessary extension to the language is merely to allow for composition of functions to handle implicit arguments appropriately. If the program in question does take an input, we argue that it is now equivalent to a function that takes the input and produces a function that takes context and returns the output. This is consistent with our view of context-aware computations as functions. The cornerstone of our approach lies in ensuring that this composition is done in a sound way. Given that types are the classical solution for providing soundness guarantees over languages, we introduce a new type for these special functions. The type of a context-aware computation is denoted by $A \downarrow^c$, where A is the type of the output and c is a set containing the types of all the contextual information it uses.

This type has interesting consequences for composition. Not only does the same input need to be available to both computations, but if one computation declares that it is interested

in a set of contextual items c_1, and the other declares that is is interested in a different set c_2, the global knowledge base that the program provides access to should provide access to contextual information from the set $c_1 \cup c_2$. Thus, we can imagine a new "composition" operator $\circ_C$ that would compose context-aware computations in the natural way:

$$\circ_C : (B \to C) \downarrow^{c_1} \times (A \to B) \downarrow^{c_2} \to (A \to C) \downarrow^{c_1 \cup c_2}$$

In other words, both the context input and the actual input correspond to inputs to our program and the dependency of the output on them can be regarded as a function. However, composition behaves differently when composing dependencies that are implicit and explicit. This idea constitutes the core of our approach. The next sections describe a practical framework based on this idea, which uses this representation for context-awareness and develops a type system for context-aware computations. This type system allows the compiler to statically verify context-aware programs for sound composition.

4.3 A prototypical type system

We can now make the previous model more concrete by analysing how we can modify a lambda calculus to incorporate these ideas. In order to do so, let us revisit the simply typed lambda calculus (STLC), which we have extended with product and unit types, in Figure 4.6.

Function composition is typically represented as a derived term:

$$f \circ g = \lambda x.f\,(g\,x)$$

As described, we intend to represent context-aware computations as functions, and define a new meaning for composition. We can do that by defining a modified lambda calculus, as well as a translation of those constructs to the STLC. We modify the previously defined lambda calculus to form the $\lambda_{\downarrow}$ system.

Our type system judgements will have the shape $\Gamma \vdash e : t \downarrow c$, stating that under environment Γ, expression e has type t, when evaluated under an environment providing the contextual information in the set c.

The notion of composability we described is here encapsulated in the application rule. We can however relate function composition and function application, if we think, as before, of constants as functions ignoring the input. In that case, in the application term $e_1\,e_2$, we can instead turn the argument into a function by writing a function that merely ignores its

$e \in Expr ::= x \mid \lambda x.e \mid e\,e \mid (e,e) \mid \pi_1 e \mid \pi_2 e \mid *$
$v \in Val ::= \lambda x.e$
$t \in Ty ::= X \mid t \to t \mid t \times t \mid 1$

$\boxed{\Gamma \vdash e : t}$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t}\ \textsc{Ax}$$

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x.e : t \to t'}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash e\,e' : t'}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ \textsc{Prod}$$

$$\frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1}\ \textsc{Fst}$$

$$\frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_2 e : t_2}\ \textsc{Snd}$$

$$\frac{}{\Gamma \vdash * : 1}\ \textsc{Unit}$$

$\boxed{e \longrightarrow e'}$

$$\frac{}{(\lambda x.e)\,e' \longrightarrow [x := e']e}\ \textsc{Beta}$$

$$\frac{}{\pi_1(e_1, e_2) \longrightarrow e_1}\ \textsc{PFst}$$

$$\frac{}{\pi_2(e_1, e_2) \longrightarrow e_2}\ \textsc{PSnd}$$

$$\frac{e_1 \longrightarrow e_2}{e\,e_1 \longrightarrow e\,e_2}\ \textsc{AppArg}$$

$$\frac{e_1 \longrightarrow e_2}{e_1\,e \longrightarrow e_2\,e}\ \textsc{AppFun}$$

$$\frac{e \longrightarrow e'}{\lambda x.e \longrightarrow \lambda x.e'}\ \textsc{LamCong}$$

Figure 4.6: Type system and Operational Semantics for the STLC.

argument: $\lambda x.e_2$, where x does not occur (freely) in e_2. If we do this, we get the equivalence (in STLC):

$$e_1 e_2 =_\beta e_1 \circ (\lambda x.e_2), x \textbf{ fresh}$$

This equivalence, paired with the previous informal definition for $\circ_C$ suggests the type system rule for application terms, CApp:

$$\frac{\Gamma \vdash e_1 : t \to t' \downarrow c \quad \Gamma \vdash e_2 : t \downarrow c'}{\Gamma \vdash e_1\, e_2 : t' \downarrow c \cup c'} \text{ CApp}$$

The remaining rules should merely propagate the contextual dependencies in a similar way, and are omitted. Note that we have not specified how to introduce or discharge contextual dependencies so this system is only useful as a skeleton to illustrate the context propagation ideas.

It is straightforward to adapt the CApp rule in order to use products to stand for unions. We can do this by representing $c \cup c'$ as $c \times c'$. If we then use types to specify contexts this corresponds to values of the form (e, e'), where e and e' would have the types corresponding to c and c' respectively. This is shown in Appendix A. However, (syntactical) products aren't associative, which implies that the collected contextual dependencies could end up being bracketed arbitrarily. This is problematic if we attempt to resolve values depending on those types of contexts, as we will have to provide, not only the appropriate values in the right order, but also bracketed in the right way. For example, if we have a computation that depends on $(Location \times Time) \times User$, and we provide it with the context $(london, (noon, user1))$, we would get a type error. We will see how to avoid the associativity problem in the next Chapter, by exploiting type classes in Haskell to preserve a canonical form for contexts. We will now define operational semantics directly by abstracting away context provision. This is useful to examine the metatheoretical implications of context-awareness in the type system, and will be the basis of the type system in Chapter 6.

4.4 Operational semantics

In this section we will present a direct semantics for $\lambda_\downarrow$. A direct semantics is very useful in exploring a more natural embedding, and its metatheoretical properties. It is also useful in designing and implementing a high-level language based on the calculus. It will become essential in chapter 6, when we attempt to extend the calculus with context-dependent types.

Here, we again extend the calculus to specify a context provision operator, forming the $\lambda_{\downarrow[]}$ calculus.

$e \in CDExpr ::= x \mid \lambda x.e \mid e\,e \mid ?l \mid l[e]$

$v \in Val ::= \lambda x.e \mid ?l$

$\boxed{\Gamma \vdash e : t \downarrow c}$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t \downarrow \varnothing}\ \textsc{CAx}$$

$$\frac{\Gamma, x : t \vdash e : t' \downarrow c}{\Gamma \vdash \lambda x.e : t \to t' \downarrow c}\ \textsc{CLam}$$

$$\frac{\Gamma \vdash e_1 : t \to t' \downarrow c \quad \Gamma \vdash e_2 : t \downarrow c'}{\Gamma \vdash e_1\,e_2 : t' \downarrow c \cup c'}\ \textsc{CApp}$$

$$\frac{l : t \in \Delta}{\Gamma \vdash ?l : t \downarrow \{l\}}\ \textsc{CLabel}$$

$$\frac{\Gamma \vdash e : t \downarrow c}{\Gamma \vdash l[e] : t \downarrow c}\ \textsc{CCTrans}$$

$\boxed{e, \Theta \longrightarrow e', \Theta'}$

$$\frac{}{(\lambda x.e)\,e', \Theta \longrightarrow [x := e']e, \Theta}\ \textsc{CBeta}$$

$$\frac{\langle v, \Theta' \rangle = pop(\Theta, l)}{l[e], \Theta \longrightarrow [?l := v]e, \Theta'}\ \textsc{CKappa}$$

$$\frac{e_1, \Theta \longrightarrow e_2, \Theta'}{e\,e_1, \Theta \longrightarrow e\,e_2, \Theta'}\ \textsc{CAppArg}$$

$$\frac{e_1, \Theta \longrightarrow e_2, \Theta'}{e_1\,v, \Theta \longrightarrow e_2\,v, \Theta'}\ \textsc{CAppFun}$$

We separate contextual identifiers from types, as we are no longer constrained by the need for embedding the type system. We call these identifiers labels, and use the metavariable l, to range over the set of all labels: *Label*. Label typings are stored in an environment Δ which, for convenience, is global. We reify the contextual values into an oracle $\Theta \in Label \to Stream(e)$, which maps labels to infinite sequences of values (representing all the values that will be read during execution). We define a scoping term $l[e]$, which replaces all occurrences of the context

label ?l in term e by the next sensed value for that label. This is also why there is no rule to evaluate ?l. The ?l construct should never appear at the top level, and it is indeed a stuck term. The oracle reifies contextual values, and is a map of type $Label \to Stream(v)$, mapping labels to infinite sequences of values (representing all the values that will be read during execution). Beta reduction is standard (CBeta). Whenever we read a value with the syntax $l[e]$ (CKappa), we remove it from the sequence, which models a change in the environment (between that point and the next read). We pop a value from the stream of values corresponding to that label, and replace all free occurrences of the label within e by that value ($l[e]$ binds the label l in e). For example, for $\Theta(UserLocation) = A; B; ...$ and $r = UserLocation[?UserLocation]$, we have:

$$\begin{aligned} &UserLocation[?UserLocation], \\ &(UserLocation \mapsto (A; B; \cdots), \cdots) \\ &\quad \longrightarrow A, (UserLocation \mapsto (B; ...), \cdots) \end{aligned}$$

The previous example, of a term that calculates the distance between a user and their home, assuming $\Delta = \{UserLocation : Location\}$ would then become:

$$UserLocation[((\lambda x.distance\ ?UserLocation\ x)\ loc\_home)]$$

Note that we are no longer restricted to using language types to identify contextual values, but instead use a separate set of labels, which can more adequately model the contextual domain. Moreover, we defer resolution of contexts to the oracle, but nevertheless determine their scopes using the $l[e]$ notation for consistency. We will now prove type soundness properties for this calculus.

Definition 4.1. *Well-formedness of oracles: We say that an oracle is well-formed if and only if all of the values for a given label have the type that Δ prescribes:*

$$wf(\Theta) \overset{def}{\iff} \forall l. \forall v. v \in \Theta(l) \wedge l : t \in \Delta \Rightarrow \vdash v : t \downarrow \varnothing$$

Well-formedness of environments is obviously preserved across reductions, as we only remove values from the lists, and Δ is constant.

Lemma 4.1. *Preservation of well-formed oracles: If we evaluate a term under a well-formed oracle, the resulting oracle is also well-formed;*

$$wf(\Theta) \wedge e, \Theta \longrightarrow e', \Theta' \Rightarrow wf(\Theta')$$

We can prove type soundness directly through proving progress and type preservation properties as usual:

Theorem 4.1. *Progress: Given a closed term e, with no free variables or undischarged contextual dependencies, if e is typeable, then it is either a value or is further reducible:*

$$wf(\Theta) \wedge \vdash e : t \downarrow \varnothing \Rightarrow e \in Val \vee \exists e' \Theta' . e, \Theta \longrightarrow e', \Theta'$$

Theorem 4.2. *Type Preservation: If a term e is typeable, and we reduce it one step, the resulting term is also typeable, with the same type and contextual dependencies.*

$$wf(\Theta) \wedge \Gamma \vdash e : t \downarrow c \wedge e, \Theta \longrightarrow e', \Theta' \Rightarrow \Gamma \vdash e' : t \downarrow c$$

These properties can be easily proven by induction over the structure of derivations. Timing and periodic sampling are outside the scope of this calculus, and could be implemented using side effects. Nevertheless, the algebraic properties of such extensions of the type system are interesting, but left to future work.

Regardless, the basic scheme for collecting contextual dependencies will remain the same. This scheme allows us to defer context provision to the user of one of these computations, or a runtime.

4.5 Discussion

We have started this chapter by analysing a model of context-awareness that showcases how context-awareness struggles with abstraction and modularity. We then related concepts in that model to functional programming, in particular the lambda calculus, and proposed a more concrete presentation of context-aware computations. We used this presentation to identify the modularity issues and propose a slight reimagining of the semantics of the lambda calculus to address them.

We then presented two separate systems, an embedding of those ideas into a standard lambda calculus, and a custom calculus with direct semantics. The former is useful for examining how to incorporate these semantics into existing functional programming languages, and will be analysed in detail in Chapter 5. The latter is useful for analysing possible designs for future programming languages, and the metatheoretical implications of the context-awareness extensions. It will be used in Chapter 6 as basis for defining a core calculus featuring context-dependent types, which will be extended to form a practical programming language in Chapter 7.

CHAPTER 5

Context-dependent Values

In the previous chapter we presented a conceptual model for context-aware computations within a functional programming language and presented a simple embedding into a lambda calculus. We will now present an exploration of the design space for context-dependent values within an industry-standard programming language known for its flexibility in embedded type systems, Haskell [M+10]. In order to do this, we will embed the $\lambda_{\downarrow}$ system we specified in the previous chapter, and extend it with features that we deem useful and aligned with our design goals. Along the way we will derive a practical usable framework based on these concepts.

In this chapter we present:

- A composable representation of context-aware computations that *automatically* derives the context dependencies needed at the type level (Section 5.1).
- An abstraction for knowledge bases which does not enforce any representation or reasoning procedure upon the knowledge base, over which we define all of our abstractions (Section 5.2.1).
- A *parameterized monad* [Atk09] that encapsulates adding context to a knowledge base, and statically verifies whether the required context information will be available at the call site of one of the previous context-aware computations (Section 5.2.4).
- A Haskell library that captures all of these abstractions in an *embedded domain specific language* (EDSL)

```
individual User

feature IsLocatedAt :: Location

data Shop = Shop { name :: String, location :: Location }

allShops :: [Shop]
allShops = ...

relevant Location (User ▷ IsLocatedAt) by distance

distance :: Location → Location → Double
distance = ...

nearestShops :: [Shop] ↓ { User ▷ IsLocatedAt }
nearestShops = sortC location allShops

main = loop do
  loc ← fetchLocation
  User ▶ IsLocatedAt := loc
  print (take 10 nearestShops)
```

Listing 5.1: An application example.

5.1 An Example Application

We present a simple implementation example of the declarative data-driven coding style for context-aware applications that we advocate in this chapter. The syntax for the example is that of a pure declarative context-aware language with Haskell-like syntax.

Our simple scenario is one where a user is walking home from work and wishes to pick up something to eat on the way. The user does not want the food to get cold by the time they reach their home, so they wish to know where the nearest shops to their current location are, and how far each of these shops are from their home. Code listings 5.1 and 5.2 implement the main features needed for this functionality, namely a sorted list of shops and a routine that shows the user how close the nearest shop is from home. This example shows the definition of the domain of contextual information the application is going to manipulate, the relevant data types and the context-aware computation that is intrinsic in the given specification.

We begin by defining the domain of relevant contextual information for the application. Individuals are the entities of the domain that we are concerned with, in this case the user (1). Features are the properties of the individuals that we wish to inspect and manipulate, in

this case where the user and their home are located at (3). The syntax `i ▷ f` is a type-level representation for feature f of individual i. We then define the data types that we will be manipulating in the application, namely shops (5). The connection between data and the contextual domain is provided in this case through a relevance relation. It states that locations are more relevant to the user the closer they are to them (10). We assume a data type `Location` is provided by some language library. Using the relevance relation, we sort a list of shops by contextual information, using the primitive `sortC`. In this case we are sorting the list of shops by their location field, using the applicable relevance relation with context (16). This creates a computation that is context-dependent, `nearestShops`. Its type reflects the contextual dependencies that have to be satisfied in order for its value to be computed. The type `a ↓ c` represents a value of type `a`, with contextual dependencies `c`, where `c` is a set of context types.

In order to execute this computation we need to provide it with context. The `do` keyword, similarly to Haskell, allows us to enter a sequential execution context. In this case the keyword will also provide a global knowledge base for storing and retrieving context. Usage of the knowledge base will be tracked and validated to ensure that contextual dependencies have been satisfied appropriately before context-dependent values are used. In the main loop of the application, we first fetch a location from the device's GPS (19) and add it to the knowledge base with the primitive expression $i \blacktriangleright f := v$, which allows us to assign the value of a feature f for the individual i as having value v. In this case, we are assigning the `IsLocatedAt` feature for the individual `User` as the location we have just retrieved (20). We then print the ten most relevant shops to the screen. The usage of context in `nearestShops` is statically verified by the compiler. Indeed, if we remove the line adding context to the knowledge base, we will get a compiler error specifying that the context of the type we removed is not available at the call site of `nearestShops`.

One of the main driving goals mentioned in the introduction was composability and code reuse. In that vein, we should be able to use our context-dependent list in the same way that we would use a regular list. In the final line of the example, we use the standard library function `take` on the list of shops. This function is completely independent from the context library, and has the type:

```
take :: Int → [a] → [a]
```

We can use this function for both regular lists and context-dependent lists. The application of this function to the sorted shops list will however push the contextual dependencies to the type of the return value:

```
take 2 nearestShops :: [Shop] ↓ { User ▷ IsLocatedAt }
```

```
 1 individual Home
 2
 3 distanceFromHome ::
 4   Location → Double ↓ { Home ▷ IsLocatedAt }
 5 distanceFromHome loc = distance loc (π (Home ▷ IsLocatedAt))
 6
 7 nearestShopDistanceFromHome ::
 8   Double ↓ { User ▷ IsLocatedAt, Home ▷ IsLocatedAt }
 9 nearestShopDistanceFromHome = distanceFromHome (head nearestShops)
10
11 exampleHomeDistance = loop do
12   loc ← fetchLocation
13   hloc ← askUserForHomeLocation
14   User ▶ IsLocatedAt := loc
15   Home ▶ IsLocatedAt := hloc
16   print nearestShopDistanceFromHome
```

Listing 5.2: Merging contextual information.

The example so far shows that context-aware values are first-class and can interact naturally with standard library functions. Moreover, if we were to use two contextual values in a single expression, such as a value depending on the home location and another depending on the user location, those two context dependencies would be merged appropriately. This will be seen in the next example. The primitive π is provided by the library, and allows us to manually project context from the knowledge base by type. We have used it in listing 5.2 to calculate the distance to the user's home of the closest shop to them. Note how the type of `nearestShopDistanceFromHome` (7-8) reflects the contextual dependencies that we are required to satisfy, namely, `User ▷ IsLocatedAt`, coming from `nearestShops` and `Home ▷ IsLocatedAt` coming from `distanceFromHome`. The application semantics of this language collect the contextual dependencies we use, in the type of the resulting value. This allows us to validate the state of the global knowledge base. In `exampleHomeDistance`, if we removed either line 14 or 15, we would no longer be adding necessary context to the knowledge base, and we would get a compile time error. This shows the basic behaviour that our EDSL provides. The next sections describe our implementation in GHC Haskell, along with the compromises that we had to take to conform to the host language.

5.2 HContext: A DSL for Context-Aware Programming

The application example in section 5.1 shows that there are two main facets to context-awareness. Firstly, defining computations that depend on implicit values, without breaking compos-

ability and type safety. Secondly, managing a global knowledge base of context, that can be accessed to provide context to the previous computations. We approach the former in sections 5.2.1 through 5.2.3 and the latter in section 5.2.4. All of the following definitions are written in Haskell, with liberal use of extensions provided by its flagship compiler GHC. In particular we assume `FlexibleContexts` and `TypeFamilies` as described in section 2.2, `GeneralizedNewtypeDeriving` which allows us to derive instances for newtypes, `QuasiQuotes` for convenience in defining our own syntactic sugar for type-level lists and `TypeOperators` which allows us to define operators at the type level. Note that none of these extensions affects decidability of type checking.

5.2.1 Context-aware Computations

Following the informal model we presented in the previous section, we start by representing context-aware computations as pure functions from a contextual value to the desired output. Hinting at the fact that this input is implicit, we define a new type for these functions, which is isomorphic to the basic Haskell arrow type:

```
newtype ContextF c a = ContextF {runContextF :: c → a}
  deriving (Functor, Applicative, Monad)
type a :↓ c = ContextF c a
```

Semantically, `:↓` declares that a function's argument is contextual and should be considered implicit. `runContextF` then allows us to take this context-aware value and apply it to a context to return a pure value. However, context-aware values differ from regular functions in that we want to think about them as having the type of the return value. Indeed, when applying regular functions to these values, the argument of the context-aware value should be treated as implicit and become the implicit argument of the final value returned by the application. This effect can be achieved thus:

```
apply :: (a → b) → a :↓ c → b :↓ c
apply f ca = ContextF (λc → f (runContextF ca c))
```

This definition is that of `fmap` for the `Reader` functor. Extending this behaviour to accepting multiple arguments in a curried manner leads to the definition of ⊛ from the `Applicative` instance of `Reader` [MP07]:

```
(⊛) :: (a → b) :↓ c → a :↓ c → b :↓ c
ff ⊛ fa = ContextF (λc → (ff `runContextF` c)
                         (fa `runContextF` c))
```

However, this abstraction is exceedingly restrictive in the type of context it is able to deal with, as it forces `c` to be constant. In our case, this would require the definition of a "universe" product type for context types, which is impractical. We would like the product type to be *automatically derived* as we use more and more contexts. Effectively, what we want is to *parameterise* the applicative functor so that it is able to manage context dependencies appropriately. In this vein, let us define a new operator $\circledast_\times$ which combines the contextual dependencies of both the function and the argument in a product type:

```
(⊛ₓ) :: (a → b) :↓ c1 → a :↓ c2 → b :↓ (c1 × c2)
```

This is the operator we need to implement the application semantics we outlined in section 5.1. In the next paragraphs, we will describe its implementation.

5.2.2 Application over Context-Aware Values

For a constant type `c`, the existing `Applicative` instance for `Reader` would be enough to achieve the behaviour we want. To see how we might generalise this approach to define $\circledast_\times$, let us specialize the type of $\circledast$:

```
(⊛) :: (a → b) :↓ (c1 × c2) → a :↓ (c1 × c2) → b :↓ (c1 × c2)
```

It seems that the only thing that we need to do to unify this type with that proposed for $\circledast_\times$ is to provide functions that generate this "universe" type. All we need to do is to precompose both functions with an appropriate projection function; of type $c_1 \times c_2 \to c_1$ for the first one and $c_1 \times c_2 \to c_2$ for the second one. In this way, the type of a composite computation can emerge from its components in a canonical way. In order for this scheme to apply to n-ary functions, however, we need to be able to represent and handle cartesian products effortlessly in Haskell. We will use the HList library as presented by Kiselyov et al [KLS04], which represents type-level lists as iterated products with a fixed structure, and provides utility functions and error handling. We use an extended version to obtain set semantics and operations. Other than the typical set operations we will use the `hProject` function, which allows us to retrieve subsets of context:

```
hProject :: (c1 ⊆ c2) ⇒ c2 → c1
```

For clarity we will use regular set notation for constraints and type operations in the code listings and refer the reader to the full code listings in Appendix C for details.

We can rely on precomposition with `hProject` to derive the universe type that we referred to previously. Then, we can just use the classic applicative instance for ((→) c), for all `c`, and we get the desired functionality. We can therefore generalize to get the $\circledast_\times$ operator:

```
(⊛×) :: (a → b) :↓ c1 → a :↓ c2 → b :↓ (c1 ∪ c2)
af ⊛× ax = ContextF ((runContextF af) . hProject) ⊛
           ContextF ((runContextF ax) . hProject)
```

This definition of $\circledast_\times$ has a more general principal type than the one we originally discussed, and generalizes to n-ary functions. We can also present a mapping between the "application" of an n-ary function to context-aware values and our combinators. Note that <$> is just infix `fmap`:

```
⟦ f x1 x2 .. xn ⟧ = f <$> x1 ⊛× x2 ⊛× ... ⊛× xn

evalC :: (c1 ⊆ c2) ⇒ a :↓ c1 → c2 → a
evalC ca k = ca `runContextF` hProject k

mkC1 :: (c → a) → a :↓ { c }
mkC1 f = ContextF (f . hHead)

mkC :: (c → a :↓ cs) → a :↓ (cs ∪ { c })
mkC = comb . mkC1
  where comb :: (a :↓ c1 :↓ c2) → (a :↓ (c1 ∪ c2))
        comb cca = ContextF $ λk → (cca `evalC` k) `evalC` k
```

`evalC` allows us to evaluate a context-dependent computation by providing it with the necessary context (or a superset). `mkC` and `mkC1` allow us to build context-aware computations. `mkC1` will have to be used when the return value of the function is not context dependent.

5.2.3 Abstract knowledge bases

We now turn to the issue of context representation. The abstractions that we have created clearly define semantics for context-aware values and ways to meaningfully combine them. However, we have not yet modelled access to context providers. In the sections that follow we assume that there is a language which is able to describe the full spectrum of context information that we might need. For the purposes of this thesis we assume that all context information that we retrieve is encoded in the same language. Moreover, we will assume that all context providers will use the same ontology when describing concepts. This is a very strong assumption, however solving this issue is not the focus of this thesis, and constitutes its own field of research [PGPM99]. To detach the current presentation from the previous semantics, we use a different syntax for `HProject`, `k :▷ c`, which is to be interpreted as a constraint that holds when we have a knowledge base of type `k` from which we can extract context information of type `c`, a set of context types. We also take this opportunity to add

additional structure to our context information. We provide support for individuals and features through the following type:

```
type family FeatureType a :: *
data Feat a = a := (FeatureType a)
```

We then represent individuals as singleton data types, and assign features to them with the `Feat` data type. The type family `FeatureType` allows us to embed the type system of features into that of Haskell. This is coupled with an arbitrary projection function, whose arguments serve solely as witnesses for the types corresponding to the individual/feature pair desired:

```
data individual ▷ feature = individual ▷ (Feat feature)
π :: a → f → FeatureType f :↓ { a ▷ f }
π _ _ = mkC1 $ λ(_ ▷ (_ := v)) → v
```

With these definitions, we have now implemented everything needed to produce the context-aware value `nearestShopDistanceFromHome`, which we discussed in section 5.1:

```
data User = User
data Home = Home
data IsLocatedAt = IsLocatedAt
type instance FeatureType IsLocatedAt = Location

distanceFromHome loc = distance loc <$> (π Home IsLocatedAt)
nearestShopDistanceFromHome =
  distanceFromHome <$> (location . head <$> nearestShops)
```

The only feature missing in our context representation is a notion of relevance of a piece of data for a user, given a set of contextual information. Relevance is realised as a predicate, stating whether a contextual value is relevant to the sorting of another non-contextual value. We define a restriction of this notion in order to aid the type checker, where we constrain the relation $\mathscr{R}(c, k)$, to instead be a *function*. This is represented as the associated type $\mathscr{R}$, which behaves as a type function, assigning a relevant context type to a regular type:

```
class Relevant a where
  type ℛ a :: *
  relevance :: a → ℛ a → Double
```

The `Location` example in section 5.1 would become:

```
instance Relevant Location where
  type ℛ Location = User ▷ IsLocatedAt
  relevance l1 (User ▷ (IsLocatedAt := l2)) = distance l1 l2
```

An example of this in action is the `sortC` function we introduced in section 5.1:

```
sortC :: (Relevant c) ⇒ (a → c) → [a] → [a] :↓ { ℛ c }
sortC contextfn xs =
  let sortfn c x y = compare (relevance (contextfn x) c)
                             (relevance (contextfn y) c)
  in  ContextF (λc → sortBy (sortfn . hOccurs $ c) xs)
```

5.2.4 Managing a global knowledge base

Our abstractions allow us to model context-aware computations and sources in a programming language. In order to make context truly implicit we would like to represent context as a shared knowledge base, that is populated by retrieving information from context sources and queried by context-aware computations. We should also be able to exploit all the typing information that we have been managing to make sure that this interaction is well-formed. It turns out that all of this is possible, using the formalism of parameterised monads [Atk09]. First, we combine a context-aware computation and a contextual information producer into one single abstraction, that of stateful computations, which is a straightforward parameterisation of the `State` functor available in the Haskell libraries. By using the parameterised monad corresponding to this functor [Atk09], we keep track of what information is in the knowledge base at the type level. The approach of using parameterized monads to provide static guarantees over a DSL has been used before. Sackman and Eisenbach [SE09] show how to provide security guarantees for an imperative language embedded in Haskell. Parameterised monads can be defined in Haskell as a generalisation of the `Monad` type class:

```
class PMonad m where
  return :: a → m c c a
  (>>=) :: m c1 c2 a → (a → m c2 c3 b) → m c1 c3 b
```

GHC's support for rebindable syntax allows us to recover `do` notation for parameterized monads. Qualified importing of libraries may be used where traditional monadic behaviour is desired. The types for the parameterised context monad (and monad transformer) then become:

```
newtype ContextRuntime c1 c2 a =
  CR { runContextRuntime :: c1 → (a, c2) }
newtype ContextRuntimeT m c1 c2 a =
  CRT { runContextRuntimeT :: c1 → m (a, c2) }
```

```
liftCRT :: Monad m ⇒ m a → ContextRuntimeT m c c a
```

We omit the PMonad instances and transformer combinators as they are essentially the same as those provided by the regular state monad. Note that our parameterised "monad transformer" is not a fully general parameterised monad transformer as it only works for non-parameterised monads. However, this is enough for the purpose of interacting with most monads present in the Haskell libraries. We then need to define a mapping from the parameterised applicative functor to the monad:

```
inContext :: (k ▷ cs) ⇒ ContextF cs a → ContextRuntime k k a
inContext cf = CR $ λk → (evalC cf k, k)
```

We must also provide combinators to add to and update the knowledge base, all whilst performing the required type-level updates. We define a function that operates on type-indexed products, which updates a value by type if it is in the product, and appends it otherwise, called `hUpdateAtTypeOrAppend` (the definition can be found in Appendix C). Using this, updating a context value in the knowledge base simply becomes:

```
(▶) :: HUpdateAtTypeOrAppend (i ▷ f) c1 c2
    ⇒ i → Feat f → ContextRuntime c1 c2 ()
individual ▶ feat = CR $
  λc' → ((), hUpdateAtTypeOrAppend (individual ▷ feat) c')
```

We may now add context values to the knowledge base represented by an HList. Note that because of the constraints in the type of `inContext`, we can only use the resulting computation if the required contextual information is present in the knowledge base (the type `ContextRuntime k k a` requires a starting knowledge base satisfying `k` and ensures that it will still satisfy `k` after the computation). The final step we must take before executing context-aware computations in this monad is enforcing an empty starting context, corresponding to the initial execution state of any program. Thus, we now define a set of execution functions for the parameterised monad that enforce this restriction. The naming for these was inspired by the execution functions provided for the `State` monad in the Haskell standard library.

```
runCR :: ContextRuntime HNil k a → (a, k)
runCR ca = runContextRuntime ca hNil
```

`evalCR` and `execCR` are defined as the appropriate projections from the result of `runCR`. We also define `evalCRT`, `execCRT` and `runCRT` as the transformer versions of these combinators. Thus, the only way to run a context-aware computation is to start with the empty context. The compiler will track all context dependencies, and abort with a compile-time error if they are not satisfied. This characteristic is arguably one of the most interesting features of our

EDSL, as we are able to reify into the type level the context dependencies of a particular computation, and thus statically guarantee that they will be fulfilled. This eliminates a whole class of potential bugs in context-aware applications, whereby the application attempts to use context when it is not stored in the knowledge base.

5.2.5 Automatically satisfying contextual dependencies

Given that our EDSL is targeting situations where the domain of contextual information can have a type system imposed on it, that uniquely identifies the type of contextual information, it is not too far-fetched to think of satisfying these implicit dependencies automatically. That is, we can use the mechanisms outlined in the previous sections to collect contextual dependencies from the main program, and we can also create a library that adds specific portions of contextual information to a global knowledge base by querying device-specific sensors. We can then tie both of these together automatically, through the type-directed translation mechanism provided by type classes.

To achieve this, we introduce a new type class, the instances of which specify which types of contextual information we can retrieve under the IO monad, for the device we are currently using.

```
pushC :: (Monad m) ⇒ c → ContextRuntimeT m HNil c ()
pushC c = CRT . const . M.return $ ((), c)

class Realizable c where
  realize :: a :↓ c → ContextRuntimeT IO HNil c a
  fetch :: IO c
  realize x = liftCRT fetch >>= pushC >> inContextT x
```

This allows us to completely hide context from the programmer who is using the EDSL. For example, if the programmer had a main loop and a function called in every iteration that could benefit from contextual information, this dependency could be added to the code for the function, and lifted to the top-level using the mechanisms the EDSL provides. We can then provide the necessary instances of Realizable for the device in question, and selectively import the ones corresponding to the retrieval technique we wish to use.

5.3 Evaluation

In order to test the expressive power of our EDSL we implemented two context-aware applications, showcasing both the abstraction capabilities provided by the library as well as the ease of interaction with existing code.

5.3.1 Presence Board

Implementing a presence board application that keeps track of all people that have checked into a certain context (e.g. a building) has become the canonical application example in context-aware libraries. This application is interesting because the presence information can then be used for more exciting context-aware applications, as will be seen. We assume an existing instance of `Realizable` for `Location` and an online service that can be used to match a location with the building that contains it, returning a circular area delimiting the range to be considered for that building/context:

```
locationToRange :: Location → IO (Location, Double)
```

The EDSL allows us to provide a reusable library for this functionality, retrieving the contextual information under the IO monad. Through the realizable type class, we can prepare this for easy use by a programmer who wishes to integrate this feature in their application. In our case, we simply supply an instance for `Realizable` for presence information:

```
data User = ...
users :: [User]
fetchLocationForUser :: User → IO Location
fetchUsers :: IO [User]
newtype Presence x = Presence [(x, Bool)] deriving (Show, Eq)

instance Realizable Location where ...
instance Realizable (Presence User) where
  fetch :: IO (Presence User)
  fetch = do
    location ← fetch
    us ← fetchUsers
    ls ← mapM fetchLocationForUser us
    (l,d) ← locationToRange location
    return . Presence $ zip us (map ((<d) . distance l) ls)
```

Using these definitions, we can develop the application code easily:

```
displayPresence :: IO () :↓ { Presence User }
displayPresence = mkC1 $ λpresence → do -- ...
main = forever (realize displayPresence)
```

Note how the programmer writing the previous code did not need to worry about how to retrieve the presence information, as it was abstracted away into a library. Then, retrieving this contextual information from the point of view of the final presence board application is simply a matter of using it at the right type, and making it implicit, using the liftings.

5.3.2 Mailing List

In order to ascertain how easy it would be to add context-awareness to an existing application, we took one of the examples used by the Context Toolkit [DAS01]: a context-aware mailing list application. This application should forward emails to only those subscribers that are located in the specific context that the mailing list applies to, in our case, physically located in a building. We located a mailing list manager application implemented in Haskell, Mhailist, publicly available on the Hackage package database [SK10]. We then proceeded to implement this behaviour without using any EDSL for implicit information. At a high level this change corresponds to retrieving presence information for the mailing list subscribers and selectively forwarding emails depending on it.

```
...
(addressees, msg)   ← return $
    case action of
      SendToList  → (addresses, addHeader listIDHeader message)
...
main = do result ← runErrorT processMessage
...
```

The modification is fairly simple, we just have to pass in the presence information to the forwarding function, and calculate it in the main loop. However, this simple change implies adding an explicit argument at every call site of the forwarding function, all the way up to the main loop. This can result in fairly significant changes to the main program. Using the existent implicit arguments feature present in GHC, we are able to propagate this dependency in a more implicit way. However, we then need to satisfy these dependencies by name, and it would be rather hard to provide an EDSL that extracts from the implicit dependencies of a computation the exact fetching routine the program should undertake, as these are iden-

tified by name. Using types to identify implicit arguments however, we are able to do just that. We can, as before, propagate the implicit argument to the main loop in an easy way. Then, in order to satisfy the main loop's context requirements, we just need to call realize, and the Realizable type class will handle retrieving the appropriate contextual information for the device and supplying it to the computation. We need to introduce the contextual dependency at the top level instead of using the lifting mechanisms presented, as otherwise we would have to fully desugar the do-notation and lift the binds. We also had to import the parameterized monad bind operator qualified as `PM.>>=` to allow us to use both monadic semantics.

```
mkC1 $ λpresence → do
...
  (addressees, msg)   ← return $
      case action of
        SendToList  →
          (filter ((isJust . flip lookup $ presence) addresses)
          , addHeader listIDHeader message)
...
main = evalCRT $ realize processMessage PM.>>= λpmsg →
          liftCRT $ do result ← runErrorT pmsg
...
```

5.4 The preprocessor

In order to achieve the syntax I outlined in the example application in the beginning of this chapter, I developed a preprocessor for the Haskell extensions we used, producing Haskell code. It begins by prefixing the source program with:

```
{-# LANGUAGE QuasiQuotes, TypeOperators,
TypeFamilies, NoImplicitPrelude, RebindableSyntax #-}
```

The following transformations are then applied to every block in the original program. We omit the necessary uppercasing/lowercasing of identifiers to conform to the Haskell requirements for identifiers, as well as all the cases where the terms would remain unchanged or where we would just recurse on subterms. This keeps the presentation concise when defining operations over Haskell's rather large syntax. We thus define $[\![\cdot]\!]_b$ for turning the top-level declarations we introduce into Haskell.

$$
\begin{aligned}
[\![\mathit{individual}\ i]\!]_b &= \texttt{data i = i} \\
[\![\mathit{feature}\ f :: t]\!]_b &= \left\{ \begin{array}{l} \texttt{data f = f} \\ \texttt{type instance FeatureType f = t} \end{array} \right\} \\
[\![\mathit{relevant}\ r\ t\ \mathit{by}\ e]\!]_b &= \left\{ \begin{array}{l} \texttt{instance Relevant r where} \\ \quad \texttt{type RelevantK r = t} \\ \quad \texttt{relevance = e} \end{array} \right\}
\end{aligned}
$$

We then apply the following transformation, $[\![\cdot]\!]_e$, on every value-level expression in the Haskell program, recursively (again omitting all cases where the terms would remain unchanged or we would just recurse on subterms):

$$
\begin{aligned}
[\![e_1\ e_2]\!]_e &= \left\{ \begin{array}{ll} [\![e_1]\!]_e\ [\![e_2]\!]_e & , e_1 \in \mathit{Exceptions} \\ \texttt{app}\ [\![e_1]\!]_e\ [\![e_2]\!]_e & , \mathit{otherwise} \end{array} \right. \\
[\![e_1 \oplus e_2]\!]_e &= \texttt{app (app (}\oplus\texttt{)}\ [\![\texttt{e}_1]\!]_\texttt{e})\ [\![\texttt{e}_2]\!]_\texttt{e} \\
\\
\mathit{Exceptions} &= \{\mathit{mkC}, \mathit{mkC1}, \mathit{return}, \mathit{unC}, \mathit{inContext}, \mathit{inContextT}, \mathit{liftCRT}\}
\end{aligned}
$$

Note that $\oplus$ stands for any Haskell operator. In our library, `app` is then defined as:

```
class App fn x r | fn x → r where
  app :: fn → x → r

instance (HUnion c1 c2 cr, cr : c1, cr : c2, a ~ a', b ~ b')
         ⇒ App ((a → b) :↓ c1) (a' :↓ c2) (b' :↓ cr) where
  app = (⊛ₓ)

instance (a ~ a', b ~ b') ⇒ App (a → b) (a' :↓ c) (b' :↓ c) where
  app = (<$>)

instance (a ~ a', b ~ b') ⇒ App (a → b) a' b' where
  app = ($)
```

This combinator translates as $\circledast_\times$ when we know that both arguments are context-dependent, `<$>` or equivalently `fmap` when only the second argument is context-dependent, and regular application (`$`) when neither arguments are context-dependent. This provides us with the desired behaviour for application. Note that this requires us to relax the instance restrictions with incoherent instances, which as we mentioned in section 2.2 has issues with type checking and different instance choices being made depending on the instantiations of `fn`, `x` and `r`. This causes no problems as long as we apply `app` to arguments of monomorphic types, as is the case in our examples. Otherwise, it can require the developer to specify constraints guiding instance resolution.

As far as types go, we need to encode the set notation we have used throughout as Haskell constraints. As we liberally used it within types, we need to collect constraints as we recurse through the types (omitting cases as before):

$$
\begin{aligned}
[\![]\!]_t &: t \to (t, k) \\
[\![t_1 \to t_2]\!]_t &= (t_1' \to t_2', (k1, k2)), \\
&\quad \textbf{where } (t1', k1) = [\![t_1]\!] \textbf{ and } (t2', k2) = [\![t_2]\!] \\
[\![t_1 \cup t_2]\!]_t &= (t_r, (\texttt{HUnion}\ \texttt{t}_1'\ \texttt{t}_2'\ \texttt{t}_\texttt{r})), \\
&\quad \textbf{where } (t1', k1) = [\![t_1]\!] \textbf{ and } (t2', k2) = [\![t_2]\!] \textbf{ and } t_r \textbf{ is globally fresh}
\end{aligned}
$$

We again apply this to all types present in the Haskell program, and merge the constraints with the calculated ones.

This preprocessor definition matches the intuition we had for the notation. Since it is mostly syntactic sugar, we do not state any properties for the translation. We have implemented the preprocessor in Haskell, thus defining a practical usable DSL for context-dependent values. The code corresponding to the example in Listing 5.1 is then:

```
data User = User

data IsLocatedAt = IsLocatedAt
type instance FeatureType IsLocatedAt = Location

data Shop = Shop{name :: String, location :: Location}
          deriving (Show, Eq)

instance Relevant Location where
```

```
  type ℛ Location = User ▷ IsLocatedAt
  relevance = distance

nearestShops :: [Shop] ↓: [h| User ▷ IsLocatedAt |]
nearestShops = app (app sortC location) allShops

main
  = app loop
      (do loc ← fetchLocation
          app (app () User) (app (app (: =) IsLocatedAt) loc)
          mcfToCrT (app print (app (app take 10) nearestShops)))
```

If we resolve the app type class in all occurences we get:

```
data User = User

data IsLocatedAt = IsLocatedAt
type instance FeatureType IsLocatedAt = Location

data Shop = Shop{name :: String, location :: Location}
          deriving (Show, Eq)

instance Relevant Location where
  type ℛ Location = User ▷ IsLocatedAt
  relevance = distance

nearestShops :: [Shop] ↓: [h| User ▷ IsLocatedAt |]
nearestShops = sortC location allShops

main
  = loop
      (do loc ← fetchLocation
          User  IsLocatedAt : = loc
          mcfToCrT (print <$> (take 10 <$> nearestShops)))
```

This corresponds to how the program would be naturally represented in the Haskell EDSL. We will present a different language in Chapter 7, which has some similarities in terms of syntax, but does not compile to Haskell. In order to highlight the differences, we will revisit this example in that Chapter.

5.5 Discussion

We have started this chapter with the goal of defining a language under which it would be natural to express context-aware applications. We introduced an example application for a hypothetical language, and throughout the chapter implemented all the necessary abstractions. It is interesting to review the example in light of the design goals, in particular to examine just how declarative a language we were able to achieve. We can start off by reformulating the example application, by just discerning what could be provided as reusable library functionality and what is actually application code. On the knowledge representation side, the application relied on the notion of an individual and their location. We also stated that other entities are more relevant to the individual the closer they are to them. These seem like generic properties, pertaining to the domain of contextual information we are interested in. Thus, in the spirit of the *separate reusable definition of ontology/typing context runtime* design goal, we are able to abstract away the following definitions:

```
individual User

feature IsLocatedAt :: Location

relevant Location (User ▷ IsLocatedAt) by distance
```

This could be placed in its own separate module, and specifies the information we are interested in, as well as some semantic relations between the individuals specified.

Next up, we should have a context runtime for the device at hand, specifying how to fetch location information using the particular set of sensors the device posesses. This is achieved by defining an instance of Realizable for the user's location:

```
instance Realizable (User ▷ IsLocatedAt) where
...
```

Again, this could be abstracted away as its own library, for the device at hand. This leaves the following definitions for the application code:

```
allShops = ...
nearestShops = sortC location allShops

main = forever (realize (print (take 10 nearestShops)))
```

This is an extremely concise formulation of the logic of a shop listing application, which is very close to how one might specify the application itself. The importance of the design

goals is made really clear here. First of all, *composability* is essential in order to apply the standard library functions `print` and `take` to the context-dependent value `nearestShops`. Secondly, *type inference* makes it so that the developer does not have to concern themself with determining the type of `nearestShops`, which as we stated is:

```
nearestShops :: [Shop] :↓ { User ▷ IsLocatedAt }
nearestShops = sortC location allShops
```

Moreover, because of the usage of HList, and the `Realizable` type class, the order in which the contextual values are provided and the order in which they are requested need not match. We thus have what we named *equality/equivalence for context equivalent types*, in the sense that values of context equivalent types are equivalent with respect to the `evalC` function. We retain type safety, as the compiler would detect the usage of a context-dependent value in a place where it was expecting a context-independent value. We believe that this example really concretizes the vision we had for the design goals, and makes the advantages of our approach clear.

The only design goal that we are missing in this language is context-dependent types. As this has implications in the type system of the language, in the next chapter we will use the $\lambda_{\downarrow[]}$ system defined in Chapter 4 and extend it with context-dependent types.

CHAPTER 6

Context-dependent Types

While the EDSL presented in the previous section already addresses some of our design goals in implementing typical context-aware applications in a concise and safe manner, it does not allow contexts to influence types as outlined in the design goals. In this section, we take a more fundamental language-based approach to embedding context in a robust and safe way whilst allowing for dynamic adaptability to involve types. To demonstrate our ideas we design a calculus (λ_{env}) that allows modes of dynamic adaptability to be specified without constraining the types of the alternatives. We then develop a type system that ensures that usage of context is not going to cause any runtime errors, due to types not matching. In order to support practical use of the ideas in a high-level language we will then develop a type inference algorithm and discuss a polymorphic extension.

In this Chapter we present:

- An extension of $\lambda_{\downarrow[]}$ to add modal context-dependent types, named λ_{env}, with a modified type system, direct operational semantics and a proof of type soundness.
- A syntax-directed version of the λ_{env} type system, allowing us to define a Hindley-Milner style type inference algorithm that is both sound and (we conjecture) complete.
- A discussion of the inclusion of simple polymorphism in the language, and the modifications that this would entail in both the type system and inference algorithm.

6.1 Motivating example

Cross-layer optimisation There are many examples of dynamical systems that are driven by context awareness. One recent example where we can see the current models of abstraction

breaking down is with the move to cross-layer optimization in wireless networking research [FGA08]. In traditional networks, code is organised according to a strict stack of layers, where for each layer we specify a well defined interface for that layer to be called with. However, this strict stack has proven too restrictive. It is becoming more common to optimise a protocol's operation using context (state and performance metrics) originating from other layers, breaking the notion of strong encapsulation with well specified interfaces.

To illustrate the power of our approach we will use cross-layer optimisation as an example. We implement a hypothetical network stack where we want to change the *physical layer* and *data layer* protocols depending on the neighbourhood density. That is, the protocols have different behaviours depending on how many neighbouring nodes each node can communicate with. This is desirable, and realistic, as certain protocols cease to function optimally under very dense networks. For instance, if a significant number of nodes share a medium (e.g. radio space) they will cause packet collisions which in turn means that additional processing such as error correction or message retransmission is required. We should note that in this example we are only concerned with local adaptation, from the point of view of a single node. Ensuring that both sender and receiver are working under the same protocol is left to a network-level protocol.

We thus assume that all the nodes in the network will implement both sparse network and dense network functionality. For this problem domain, the return type of the physical layer will vary depending on the network density. One can imagine the data layer protocol accessing the quality of the links between it and its neighbours to make routing decisions. If the network is sparse then a detailed mathematical representation of link quality can be used. However, the system would become sluggish in denser networks [BKM+12]. Here a notion of "good", "ok", and "bad" link quality thresholds would suffice.

Using current approaches, this variability is something that is usually handled with either common supertypes (potentially paired with dangerous downcasts) or tagged unions (which increase the amount of bookkeeping the programmer needs to do, usually with an associated runtime cost).

Cross-layer optimisation in λ_{env} The first thing that we want to be able to do is retrieve contextual values. We do so as in $\lambda_{\downarrow[]}$, by querying context labels, with the term $?l$, where l is the desired label. For example, to query the value of the neighbourhood density, we would use the term $?NeighbourhoodDensity$. We will then assign terms in the calculus types which reflect their contextual dependency. The value of a label query is indistinguishable from a non-contextual value, which makes it easy to apply functions to it and process it.

Our system also supports the definition of context-dependent types, whereby the type can depend on the contextual value queried. This is achieved by using terms of the form:

$$\textbf{ccase}\ l\ \textbf{of}\ \{v_1 \Rightarrow e_1; ..; v_n \Rightarrow e_n\}$$

In this term we declare an alternative e_i for every possible value v_i of the contextual label l. The types for these terms are then of the form:

$$\textbf{Case}\ l\ \textbf{of}\ \{v_1 \Rightarrow t_1; ..; v_n \Rightarrow t_n\}$$

In order to express variability in terms of the neighbourhood density, we change the return type of both layers to reflect this variability, in effect creating a conditional protocol for the layer. For clarity, in this section we relax our notation convention and allow arbitrary names for variables relying on the descriptions to disambiguate when necessary. Let us then consider values corresponding to two different packets, of different types, one for each protocol:

$$packet_1 : P_1 \text{ and } packet_2 : P_2$$

We can combine them in the following term in our calculus:

$$\begin{aligned} phy = \ & \textbf{ccase}\ NeighbourhoodDensity\ \textbf{of} \\ & \{Low \Rightarrow packet_1; High \Rightarrow packet_2\} \end{aligned}$$

This term inspects the contextual $NeighbourhoodDensity$ and depending on whether it is Low or $High$, returns $packet_1$ or $packet_2$, which have types P_1 and P_2, respectively. The type reflects this dependency explicitly. This example shows how to access a global value (neighbourhood density) and provide alternatives of different types depending on the value read. $packet_1$ has type P_1, $packet_2$ has type P_2 and the physical layer will then have the following type:

$$phy : \ \textbf{Case}\ NeighbourhoodDensity\ \textbf{of}\ \{Low \Rightarrow P_1; High \Rightarrow P_2\}$$

This type means that *phy* will return a value of type P_2 in a dense network, but will switch to a value of type P_1 when the neighbourhood density is low. To illustrate composability, we can then define two data layers, with the following types/protocols. Note that we always use variables in this section for brevity but these alternatives would usually be more complex expressions.

$$data_1 : P_1 \rightarrow D_1 \text{ and } data_2 : P_2 \rightarrow D_2$$

We can then compose them both under the following conditional protocol, as before:

$$data = \textbf{ccase}\ NeighbourhoodDensity\ \textbf{of} \\ \{Low \Rightarrow data_1; High \Rightarrow data_2\}$$

Which in our system will have the type:

$$data : \textbf{Case}\ NeighbourhoodDensity\ \textbf{of} \\ \{Low \Rightarrow P_1 \rightarrow D_1; High \Rightarrow P_2 \rightarrow D_2\}$$

We can then apply this function to the result of the physical layer without any syntactic noise:

$$data\ phy : \textbf{Case}\ NeighbourhoodDensity\ \textbf{of} \\ \{Low \Rightarrow D_1; High \Rightarrow D_2\}$$

Application terms are only typeable if the alternatives match. For instance, if we consider the term $id_{p1} : P_1 \rightarrow P_1$, corresponding to the identity function for P_1, we have that the term $id_{p1}\ phy$ is not typeable, as we do not provide function alternatives for all the argument alternatives. However, we would also like to provide a context-invariant application layer, with the type:

$$app : D \rightarrow A$$

This application layer expects a certain uniformised data packet of type D. We can now present a merging term, that distills the result of both protocols into a unified common type (thus disallowing the application layer from adapting to the protocol which was used underneath without breaching the abstraction). We can define a term which brings both alternatives into this common type:

$$data\_merge : \textbf{Case}\ NeighbourhoodDensity\ \textbf{of} \\ \{Low \Rightarrow D_1 \rightarrow D; High \Rightarrow D_2 \rightarrow D\}$$

Our final stack can then be defined as follows:

$$app\_stack : A \\ app\_stack = app\ (data\_merge\ (data\ phy))$$

In order to run this stack we then need to provide the neighbourhood density value, by retrieving it with the device's sensors (outside of the scope of this calculus), and discharging the context obligation. We do so with the following syntax:

$$NeighbourhoodDensity[app\_stack]$$

Thus binding all the free labels and making the term executable.

6.2 The λ_{env} calculus

6.2.1 Syntax

Our system, λ_{env}, is an extension of $\lambda_{\downarrow[]}$ which we presented in section 4.4. Recall that expressions of the form $l[e]$ read label l and substitute l by the next value in the oracle for the label, in all occurences of l within e. It is particularly important to delimit context scopes in the presence of context-dependent types in order to ensure consistency of the alternative types, as we will see. As before, $?l$ asks for the last read value of l and does not read a new value. A new term is added, *ccase*, which allows us to perform a dependently typed case analysis on the context. This term is used in order to introduce context-dependent types.

We stratify types in two tiers: simple types t, which are the usual types from the simply typed lambda calculus, and context-dependent types T. This is done in order to ensure that the alternatives always appear at the top level of a type, and more importantly, that they never appear in the contravariant position of a function type. In this way the type: $\textbf{Case}\, l\, \textbf{of}\, \{...\} \rightarrow T$ is forbidden. This makes it simpler to define a more syntax-directed type system, as we can handle the alternatives in a uniform way. It also does not have a significant effect on expressivity as we will see when we examine the type system in more detail.

6.2.2 Operational semantics

The operational semantics are represented in Figure 6.2 and have the format $r, \Theta \longrightarrow r, \Theta$ where r is a runtime expression and as before, Θ is an oracle that represents the environment. Runtime expressions extend expressions with a tagged *ccase* construct that contains both the label and the value it has at that particular point in the execution. Whenever we substitute a label in a *ccase* expression we need to tag it with the label that we substituted it by. This helps us to prove soundness through type preservation, but is ghost state not necessary for execution. An example of the usage of the oracle is:

$e \in Expr ::= x \mid \lambda x.e' \mid e\,e' \mid ?l \mid \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow e_i}^{\,i}\} \mid l[e]$
$r \in RTExpr ::= x \mid \lambda x.r \mid r\,r' \mid ?l \mid \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{r_i \Rightarrow r'_i}^{\,i}\} \mid l[r] \mid \mathbf{ccase}\, l\, r\, \mathbf{of}\, \{\overline{r_i \Rightarrow r'_i}^{\,i}\}$
$v \in Val ::= \lambda x.r \mid ?l \mid \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow r_i}^{\,i}\}$
$T \in Alt ::= t \mid \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i}^{\,i}\}$
$t \in Ty ::= X \mid t \to t'$
$l \in Label$
$\Gamma \in Var \rightharpoonup Ty$
$\Delta \in Label \to Ty$
$\Theta \in Label \to Stream(RTExpr)$
$c \in \mathscr{P}(Label)$

Figure 6.1: Syntax of λ_{env}.

$$
\begin{aligned}
&Location[\mathbf{ccase} Location\ \mathbf{of}\ \{\cdots\}],\\
&(Location \mapsto (A; B; \cdots), \cdots)\\
&\quad \longrightarrow\ \mathbf{ccase}\ Location\ A\ \ \mathbf{of}\ \{\cdots\}, (Location \mapsto (B; \ldots), \cdots)
\end{aligned}
$$

CCase selects a case from several alternatives, and is straightforward. The remaining rules are congruence rules as usual for the call by value lambda calculus. We need to enforce an evaluation strategy as the unrestricted calculus is not confluent. For instance, if we allowed evaluation to occur on both sides of an application in any order, the expression $l[\mathbf{ccase}\, l\, \mathbf{of}\, \{e_1 \Rightarrow x;\ e_2 \Rightarrow y\}]\, l[z]$, would have two normal forms, when evaluated under a Θ such that $\Theta(l) = e_1; e_2; ..$: we could reduce to $x\,z$ if the first $l[..]$ is evaluated first and $y\,z$ when otherwise. Thus, within an application term we first fully evaluate the right hand side to a value with rule AppArg, and then evaluate the left hand side with rule AppFun.

6.2.3 Type system

Label typings are stored in an environment Δ which, for convenience, is global. All of the types in Δ must be ground. The judgements of the type system are of the form $\Gamma \vdash e : T\,!\,c$, and mean that under the (usual) variable environment Γ expression e has type T and will read new values for the labels in the set c. Tracking this set is important because we need to ensure that within an application expression $e_1\,e_2$, both e_1 and e_2 will use the same oracle

$$r \in RTExpr ::= x \mid \lambda x.r \mid r\, r' \mid ?l \mid \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{r_i \Rightarrow r_i'}^{\,i}\} \mid l[r] \mid \mathbf{ccase}\, l\, r\, \mathbf{of}\, \{\overline{r_i \Rightarrow r_i'}^{\,i}\}$$

$\boxed{r, \Theta \longrightarrow r', \Theta'}$

$$\frac{}{(\lambda x.r)\, r', \Theta \longrightarrow [x := r']r, \Theta} \quad \textsc{Beta}$$

$$\frac{\langle v, \Theta' \rangle = pop(\Theta, l)}{l[r], \Theta \longrightarrow [l := v]r, \Theta'} \quad \textsc{Kappa}$$

$$\frac{v = v_i}{\mathbf{ccase}\, l\, v\, \mathbf{of}\, \{\overline{v_i \Rightarrow r_i}^{\,i}\}, \Theta \longrightarrow r_i, \Theta} \quad \textsc{CCase}$$

$$\frac{r_1, \Theta \longrightarrow r_2, \Theta'}{r\, r_1, \Theta \longrightarrow r\, r_2, \Theta'} \quad \textsc{AppArg}$$

$$\frac{r_1, \Theta \longrightarrow r_2, \Theta'}{r_1\, v, \Theta \longrightarrow r_2\, v, \Theta'} \quad \textsc{AppFun}$$

$$pop(\Theta, l) = \langle v, \Theta[l \mapsto \overline{v}] \rangle \text{ if } \Theta(l) = v; \overline{v}$$

Substitution is as usual except for:

$$[l := v'](\mathbf{ccase}\ \ l'\ of\ \{v_1 \Rightarrow r_1; ..; v_n \Rightarrow r_n\}) =$$
$$\quad \mathbf{ccase}\, l\, v'\, of\, \{v_1 \Rightarrow [l := v']r_1; ..; v_n \Rightarrow [l := v']r_n\} \text{ if } l = l'$$
$$[l := v](?l') = v \text{ if } l = l'$$

Figure 6.2: Operational Semantics of λ_{env}.

$\boxed{\Gamma \vdash e : T\,!\,c}$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t\,!\,\varnothing}\ \textsc{CAx}$$

$$\frac{\Gamma, x : t \vdash e : T\,!\,c}{\Gamma \vdash \lambda x.e : t \underline{\rightarrow} T\,!\,c}\ \textsc{CLam}$$

$$\frac{l : t \in \Delta}{\Gamma \vdash ?l : t\,!\,\varnothing}\ \textsc{CLabel}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : T\,!\,c \\ l : t \in \Delta\end{array}}{\Gamma \vdash l[e] : T\,!\,c \cup \{l\}}\ \textsc{CCTrans}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : T_1\,!\,c_1 \\ \Gamma \vdash e_2 : T_2\,!\,c_2 \\ T' = T_1 \cdot T_2 \\ compat(T_1, c_2) \\ compat(T_2, c_1)\end{array}}{\Gamma \vdash e_1\, e_2 : T'\,!\,c_1 \cup c_2}\ \textsc{CApp}$$

$$\frac{\begin{array}{l}l : t \in \Delta \\ \overline{\vdash_v v_i : t}^{\,i} \\ \overline{\Gamma \vdash e_i : T_i\,!\,c_i}^{\,i} \\ c' = \cup \overline{c_i}^{\,i} \\ exhaustive(\overline{v}, t)\end{array}}{\Gamma \vdash \mathbf{ccase}\, l\, \mathbf{of}\, \{\, \overline{v_i \Rightarrow e_i}^{\,i} \,\} : \mathbf{Case}\, l\, \mathbf{of}\, \{\, \overline{v_i \Rightarrow T_i}^{\,i} \,\}\,!\,c'}\ \textsc{CCases}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : T\,!\,c \\ T \sim T'\end{array}}{\Gamma \vdash e : T'\,!\,c}\ \textsc{CSubst}$$

Figure 6.3: Type system of λ_{env}.

$\boxed{T \cdot T = T}$

$$T \cdot T' = compute\_app(T, \varnothing, T')$$

$\boxed{compute\_app(T, lm, T') = T'''}$

$$\frac{t = (\!|T|\!)(lm)}{compute\_app(t \to t', lm, T) = t'} \quad \textsc{CASimpTy}$$

$$\frac{\overline{compute\_app(T_i, lm \cup \{(l, v_i)\}, T) = T_i'}^{\,i}}{compute\_app(\mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i}^{\,i}\}, lm, T) = \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i'}^{\,i}\}} \quad \textsc{CAAlt}$$

$\boxed{(\!|T|\!) = (\mathscr{P}(l \times v) \rightharpoonup T)}$

$$(\!|\, \mathbf{Case}\, l \text{ of } \{\overline{v \Rightarrow T}\}|\!)(lm) = t, \ \mathbf{iff}\ \exists lm'.(l, v') \notin lm' \wedge lm = lm' \cup \{(l, v_i)\} \wedge (\!|T_i|\!)(lm') = t_i$$
$$(\!|t|\!)(\varnothing) = t$$

$\boxed{T \leq T}$

$$T \leq T' \overset{def}{\iff} lm \mapsto t \in (\!|T|\!) \Rightarrow \exists lm'.lm' \mapsto t \in (\!|T'|\!) \wedge lm \subseteq lm'$$

$\boxed{T \sqsubseteq T}$

$$T \sqsubseteq T' \overset{def}{\iff} T \leq T' \wedge lm' \mapsto t \in (\!|T'|\!) \Rightarrow \exists lm \mapsto t \in (\!|T|\!) \wedge lm' \subseteq lm$$

$\boxed{T \sim T}$

$$T \sim_1 T' \overset{def}{\iff} T \sqsubseteq T' \vee T' \sqsubseteq T$$

$$T \sim T' \overset{def}{\iff} T \sim_1 T' \vee (\exists T''.T \sim T'' \wedge T'' \sim T')$$

$\boxed{canonical(T) = T}$

$$canonical(T) = T' \overset{def}{\iff} \forall T''.T'' \sqsubseteq T \Rightarrow T' \sqsubseteq T''$$

$\boxed{compat(T, c)}$

$$compat(T, c) \overset{def}{\iff} canonical(T) = T' \wedge \{l | l \in lm, lm \mapsto t \in (\!|T'|\!)\} \cap c = \varnothing$$

$\boxed{exhaustive(\overline{v}, t)}$

$$exhaustive(\overline{v}, t) \overset{def}{\iff} \forall v.\ \vdash_v v : t \Rightarrow v \in \overline{v}$$

$\boxed{t \underline{\to} T}$

$$t \underline{\to} t' = t \to t'$$
$$t \underline{\to}\, \mathbf{Case}\, l\, \mathbf{of}\, \{v_1 \Rightarrow T_1; \ldots; v_n \Rightarrow T_n\} = \mathbf{Case}\, l\, \mathbf{of}\, \{v_1 \Rightarrow t \to T_1; \ldots; v_n \Rightarrow t \to T_n\}$$

Figure 6.4: Auxiliary relations.

to determine the type of the value they will return. In addition to this, the effective type of a term should include the set of labels that will have to be read in order for that term to be executed, which corresponds to the free labels of that term. We denote effective type judgements as $\Gamma \vdash e : T \downarrow c \,!\, c'$, where $c = fl(e)$. Thus:

$$\Gamma \vdash e : T \downarrow c \,!\, c' \overset{def}{\iff} \Gamma \vdash e : T \,!\, c' \wedge fl(e) = c$$

The set of free labels is calculated as standard:

$$fl(e) = \begin{cases} \varnothing & , e = x \\ fl(e') & , e = \lambda x.e' \\ \{l\} & , e = ?l \\ fl(e_1) \cup fl(e_2) & , e = e_1\, e_2 \\ fl(e') \setminus \{l\} & , e = l[e'] \\ \{l\} \cup (\bigcup \overline{fl(e)}) & , e = \textbf{ccase}\ l\ \text{of}\{\overline{v \Rightarrow e}\} \end{cases}$$

We can make an analogy to type and effect systems if we think of one mutable variable per context label. In this way, the labels in set c correspond to the labels that will be read, whereas the labels in set c' correspond to the labels that will be written. In order to ensure consistency it suffices to keep track of the labels that will be written, so we omit the read effects from the type system. However, as we will see, read effects should be made visible to the programmer, as they determine whether a term can be reduced or not.

The type system is presented in full in Figure 6.3. The rule for typing lambda abstractions (Lam) is standard, except for the normalisation we have to do to make sure alternatives only occur at the top-level. This is done by pushing the argument type inside all of the alternatives and is achieved using the $\underline{\rightarrow}$ operator.

The Label rule merely uses the typing provided by the global Δ environment. The CTrans rule extends the set of labels that will be read. For example, $NeighbourhoodDensity[data]$ and $NeighbourhoodDensity[phy]$ have the following types:

$$NeighbourhoodDensity[data] : T_{data} \,!\, \{NeighbourhoodDensity\}$$
$$NeighbourhoodDensity[phy] : T_{phy} \,!\, \{NeighbourhoodDensity\}$$

where

$$
\begin{aligned}
T_{data} = \ & \textbf{Case}\ NeighbourhoodDensity\ \textbf{of} \\
& \{Low \Rightarrow P_1 \to D_1; High \Rightarrow P_2 \to D_2\} \\
T_{phy} = \ & \textbf{Case}\ NeighbourhoodDensity\ \textbf{of}\ \{Low \Rightarrow P_1; High \Rightarrow P_2\}
\end{aligned}
$$

The rule for function application (App) has to solve two main challenges: firstly, it needs to ensure that the function and argument will not read different values for the same label. Secondly, it needs to ensure that all the alternatives in the argument are covered by the function in the same way.

Regarding the first challenge, if we tried to apply the term $NeighbourhoodDensity[data]$ to $NeighbourhoodDensity[phy]$ from before we would get an incorrect term:

$$NeighbourhoodDensity[data]\ NeighbourhoodDensity[phy]$$

We prevent this term from typing by enforcing a *compat* check in the App rule, whereby we ensure that labels used to evaluate one side of the application are not used to determine the type of the other side. This term is "wrong" in the sense that we can't ensure that both sides of the application will be evaluated under the same context, which violates type soundness.

An example of the second challenge appeared in Section 6.1, where we would like *data phy* to be typeable, but id_{p1} *phy* not to be because it does not cover the alternative P_2 from *data*. In order to do this independently of the way/order we nest the **Case** types, we first switch to a more abstract representation that factors out the ordering. We can think of the original representation as an n-ary labelled tree, with labels at the branches, label values at the edges and types at the leaves. We switch to a representation which flattens that tree by recording a mapping from the sets of labels/value pairs to the types. An example of this flattening is illustrated in Figure 6.5.

This flattening is computed using the $(\!|\cdot|\!)$ function, defined in Figure 6.4. Using this representation it is then much easier to compute the resulting type for an application. As an example, suppose we want to type the term *data phy*. For *data* we would have the following representation:

$$
\begin{aligned}
(\!|T_{data}|\!) = (\{(NeighbourhoodDensity, Low)\} & \mapsto P_1 \to D_1, \\
\{(NeighbourhoodDensity, High)\} & \mapsto P_2 \to D_2)
\end{aligned}
$$

The $\cdot$ operator, defined in Figure 6.4 traverses the *Case* type for the function, and for every case,

$$\textbf{Case}\ l_1\ \textbf{of}\ \{e_1 \Rightarrow\ \textbf{Case}\ l_2\ \textbf{of}\ \{v_2 \Rightarrow t_1; v_3 \Rightarrow t_2\}; v_4 \Rightarrow t_3\}$$

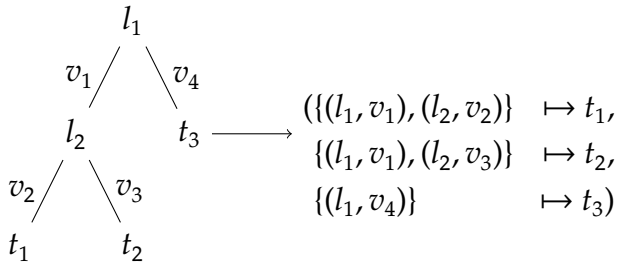

Figure 6.5: An example of the intermediate representation used for alternatives.

checks that the type of the argument that corresponds to that path is equal to the argument type of the function and returns the appropriate type for the function application.

We can then compute the resulting type, by analysing the label values for all the alternatives in the argument, checking that they yield a matching type in the representation and returning the result type. In the previous example:

$$\begin{aligned}
&T_{data} \cdot T_{phy} = compute_app(T_{data}, \varnothing, T_{phy}) \\
&compute_app(\ \textbf{Case}\ NeighbourhoodDensity\ \textbf{of} \\
&\quad \{Low \Rightarrow P_1 \to D_1; High \Rightarrow P_2 \to D_2, \varnothing, T_{phy}\}) \\
&=\ \textbf{Case}\ NeighbourhoodDensity\ \textbf{of} \\
&\quad \{Low \Rightarrow compute_app(P_1 \to D_1, \\
&\qquad\qquad \{(NeighbourhoodDensity, Low)\}, T_{phy}); \\
&\quad High \Rightarrow compute_app(P_2 \to D_2, \\
&\qquad\qquad \{(NeighbourhoodDensity, High)\}, T_{phy})\} \\
&=\ \textbf{Case}\ NeighbourhoodDensity\ \textbf{of}\ \{Low \Rightarrow D_1; High \Rightarrow D_2\}
\end{aligned}$$

Note how the use of the representation means that both the ordering and nesting of alternatives and labels is irrelevant to the calculation. The second alternative in *compute_app* is needed in case we have a context application in the argument, resulting in a constant value, as happens with the *data_merge* term. We allow the argument to contain an alternative type if this type is actually not context-dependent.

The Cases rule types all the expressions independently, collecting the sets from all the subexpressions. This rule needs to ensure that the branches provided completely cover the type in question, which is done through the *exhaustive* predicate. This implies that we can only perform dependent case analysis on finite types, that we can fully enumerate. The system is parametric on the definition of a value type system with judgements of form $\Gamma \vdash_v v : t$. This

can be, for instance, a context-independent subset of λ_{env} such as the STLC or some other system, as long as type checking is decidable.

Another problem that we wish to solve in the system is simplifying verbose types. For example, we would like to simplify the verbose type:

$$\textbf{Case}\ NeighbourhoodDensity\ \textbf{of}\ \{Low \Rightarrow A; High \Rightarrow A\}$$

As merely A. In order to do so, we define an equivalence relation, ($\sim$), whereby both the above types would be considered equivalent. To do so, we define a relation on types $\sqsubseteq$, whereby for a type T and a type T' they are related by $T' \sqsubseteq T$ if T' just removes redundant clauses from T. Following this relation, $A \sqsubseteq$ **Case** $NeighbourhoodDensity$ **of** $\{Low \Rightarrow A; High \Rightarrow A\}$. Reflexivity and transitivity of this relation are evident from the definition. In fact, types that are related by this relation are then equivalent in the sense that they only differ in redundant clauses. This allows us to define the desired equivalence relation, $\sim$, as the symmetric transitive closure of $\sqsubseteq$.

We allow types to be substituted for equivalent types with rule CSubst. Note that we need to be able to expand types as well as contract them in order to do the aforementioned representation matching in the CApp rule. For example, consider expressions e_1 and e_2, with the following types:

$$\begin{aligned} e_1 : \ & \textbf{Case}\ NeighbourhoodDensity\ \textbf{of} \\ & \quad \{Low \Rightarrow A \to B; High \Rightarrow A \to C\} \\ e_2 : \ & A \end{aligned}$$

Consider the expression $e_1\, e_2$. In this case the type of e_2 would need to expand so that it covers all the cases in the type of e_1. It would have to become:

$$e_2 : \textbf{Case}\ NeighbourhoodDensity\ \textbf{of}\ \{Low \Rightarrow A; High \Rightarrow A\}$$

Yielding the final type:

$$e_1\, e_2 : \textbf{Case}\ NeighbourhoodDensity\ \textbf{of}\ \{Low \Rightarrow B; High \Rightarrow C\}$$

The type system for runtime expressions $\vdash_r$ is similar, except that when typing tagged **ccase**

expressions, we ensure that the type of the tag and the type of the label match:

$$\frac{\begin{array}{l} l : t \in \Delta \\ \overline{\vdash_v v_i : t}^{\,i} \\ \overline{\Gamma \vdash_r e_i : T_i ! \, c_i}^{\,i} \\ \vdash_v v : t \\ c' = \cup \overline{c_i}^{\,i} \\ \mathit{exhaustive}(\overline{v}, t) \end{array}}{\Gamma \vdash_r \mathbf{ccase}\, l\, v\, \mathbf{of}\, \{\overline{v_i \Rightarrow e_i}^{\,i}\} : \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i}^{\,i}\} ! \, c'} \text{ RLCases}$$

All the other rules are analogous to the ones in the expression type system. This type system is presented in full in Appendix B.7.1.

6.2.4 Properties of the auxiliary relations

Before we can begin proving our main results, it is useful to prove some auxiliary Lemmas related to how the auxiliary relations interact with each other:

Lemma 6.1. *Preservation of $\sim$ over $\underline{\rightarrow}$: The right hand side of an $\underline{\rightarrow}$ type can be replaced by a $\sim$-equivalent type, the result being $\sim$-equivalent to the original. Moreover, this is the only way in which $\underline{\rightarrow}$ types can be $\sim$ equivalent.*

$$T \sim T' \Leftrightarrow t \underline{\rightarrow} T \sim t \underline{\rightarrow} T'$$

Proof. Obvious by definition of $\underline{\rightarrow}$. Note that as far as $(\!|\cdot|\!)$ representations go, the domain, i.e. label assignment maps ($lm \in \mathscr{P}(Label \times Val)$) will remain unaffected by $\underline{\rightarrow}$ and the corresponding types will all get affected by appending "$t \rightarrow$" to the left of the existing type. Hence all label assignment map relations are preserved, and all type equalities are also preserved. □

Lemma 6.2. *Preservation of $\sim$ over* **Case***: If we form two* **Case** *types, identical in labels and values, and with types $\sim$-equivalent pairwise, those types will also be $\sim$-equivalent:*

$$\overline{T \sim T'} \Rightarrow \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v \Rightarrow T}\} \sim \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v \Rightarrow T'}\}$$

Proof. By examining the definition of $(\!|\cdot|\!)$. Note that the representation of a **Case** type is formed by merely appending a label-value pair to the label assignment map. Thus, if the representations of all the $\overline{T}$ and $\overline{T'}$ are such that they are $\sim$-equivalent pairwise, the composite **Case** types will also be $\sim$-equivalent. In particular, note that appending the same label-value mapping pairwise, will make the subsetting portion of the $\sim$ definition hold, and the codomain of the representation is unchanged. □

Lemma 6.3. *Soundness of $\cdot$ with respect to $\underline{\rightarrow}$: If we have a type that is $\sim$-equivalent to a type constructed with $\underline{\rightarrow}$, the $\cdot$ application of this type, if defined is equal to the right-hand side of the $\underline{\rightarrow}$ and what it is applied to is $\sim$-equivalent to the left-hand side:*

$$T_1 \sim t \underline{\rightarrow} T_1' \wedge T = T_1 \cdot T_2 \Rightarrow T \sim T_1' \wedge T_2 \sim t$$

Proof. By examining the definition of $\cdot$ and $\underline{\rightarrow}$. Note that $\cdot$ is defined based on representations only and $\underline{\rightarrow}$ merely constructs an arrow type with the same argument for every type in the codomain of the representation. When checking the representation, at CASimpTy, the $\cdot$ application is only defined if the argument type of the function and the type of the argument match exactly. Hence, for the application to be defined, all the alternative types in the right hand side of the application must be the same: $T_2 \sim t$. The return type of that application is going to be the corresponding alternative in T_1'. Since we may have added or removed redundant alternatives in the formation of T_1, we know that the return type is not equal, but $\sim$-equivalent to T_1': $T \sim T_1$. □

Lemma 6.4. *A compat assertion on a type resulting from $\cdot$-application implies the same compat property on both function and argument types:*

$$T_f = T_1 \cdot T_2 \wedge compat(T_f, c) \Rightarrow compat(T_1, c) \wedge compat(T_2, c)$$

Proof. Note that for $\cdot$ to succeed, the same labels need to be used to determine the type of both the argument and the function. Thus, any restrictions that apply to the set of labels of the resulting type has to also apply to the set of labels of both sides of the application. □

6.2.5 Soundness

In order to make all the following proofs easier by induction, we will define a syntax-directed version of the type system. Note that with the exception of the CSubst rule, the type system is completely syntax-directed. The CSubst rule is only necessary for typeability when we need to match alternatives in the application rule. We can formalize this claim by defining a restricted system, $\Gamma \vdash_a r : T ! c$, where we remove the CSubst rule and instead allow substitution of types by equivalents to only happen within the CApp rule. Thus, $\vdash_a$ is similar to $\vdash$

without the CSubst rule, with the exception of a modified CApp rule:

$$\frac{\begin{array}{l}\Gamma \vdash_a r_1 : T_1 ! c_1 \\ \Gamma \vdash_a r_2 : T_2 ! c_2 \\ T' = T_1' \cdot T_2' \\ T_1' \sim T_1 \\ T_2' \sim T_2 \\ compat(T_1', c_2) \\ compat(T_2', c_1)\end{array}}{\Gamma \vdash_a r_1\, r_2 : T' ! c_1 \cup c_2} \text{ AApp}$$

We present $\vdash_a$ in full in appendix B.7.2. Soundness of $\vdash_a$ is trivial:

Theorem 6.1. *Soundness of $\vdash_a$: If a term is typeable in the syntax-directed type system with type T, it is typeable in the unrestricted system with the same type:*

$$\Gamma \vdash_a r : T ! c \Rightarrow \Gamma \vdash_r r : T ! c$$

Proof. We proceed by induction on the typing derivation. All cases where the rule from $\vdash_a$ has an equivalent rule in $\vdash_r$ are trivial. The only remaining case is AApp. We can use the Subst rule on the $\sim$ premises, to make the types match and can then simply apply App. □

Our previous claim about completeness can then be stated as:

Theorem 6.2. *Completeness of $\vdash_a$: If a term is typeable in the unrestricted system with type T, it is typeable in the restricted system with a type T', such that $T \sim T'$.*

$$\Gamma \vdash_r r : T ! c \Rightarrow \exists T'.\Gamma \vdash_a r : T' ! c \wedge T \sim T'$$

Proof. We prove this Theorem by induction on the derivation of $\Gamma \vdash_r r : T ! c$.

Case Ax $r = x$, $T = t$ and $x : t \in \Gamma$. We can choose T' to be T. We can then apply AAx to get the left side of the conjunction. The right side holds by reflexivity of $\sim$.

Case Lam $r = \lambda x.r'$ and $T = t_1 \underline{\rightarrow} T_2$. By the induction hypothesis (IH) we know that there is a T_2' such that $\Gamma, x : t_1 \vdash_a r' : T_2' ! c$ and $T_2 \sim T_2'$. We can thus pick T' to be $t_1 \underline{\rightarrow} T_2'$. We can then apply ALam to get the left hand side of the conjunction. $t_1 \underline{\rightarrow} T_2 \sim t_1 \underline{\rightarrow} T_2'$ follows by Lemma 6.1.

Case Lbl $r = ?l$ and $T = t$ with $l : t \in \Delta$. This case is trivial, as Δ remains the same, so we can just apply ALbl and reflexivity of $\sim$ to get the conclusion.

Case CTrans $r = l[r']$. This case is also trivial, as we know from the IH that there is a T'' such that $\Gamma \vdash_a r' : T''!\,c$ with $T'' \sim T$, so we can just choose T' to be T'' and apply CTrans. The $\sim$ part of the conclusion matches the one in the IH.

Case App $r = r_1 r_2$ and $T = T_1 \cdot T_2$ with $\Gamma \vdash_r r_1 : T_1!\,c_1$, $\Gamma \vdash_r r_2 : T_2!\,c_2$, $compat(T_1, c_2)$, and $compat(T_2, c_1)$. From the IH we know:

- there is a T_1' such that $\Gamma \vdash_a r_1 : T_1'!\,c_1$ and $T_1' \sim T_1$, and
- there is a T_2' such that $\Gamma \vdash_a r_2 : T_2'!\,c_2$ and $T_2' \sim T_2$.

We can choose $T' = T$, and we have everything we need to apply AApp. The $\sim$ part of the conclusion follows by reflexivity.

Cases CCases and LCases: $r =$ **ccase** l **of** $\{\overline{v \Rightarrow r}\}$ and $T =$ **Case** l **of** $\{\overline{v \Rightarrow T}\}$. From the IH we know that:

- there are $\overline{T''}$ such that
$$\overline{\Gamma \vdash_a r_i : T_i''!\,c_i}^{\,i} \wedge \overline{T'' \sim T}$$

We can thus choose T' to be **Case** l **of** $\{\overline{v \Rightarrow T''}\}$. For the right side we can apply AACases or ALCases, respectively. $T \sim T'$ follows by Lemma 6.2.

Case Subst From the premises of this case we know there is T'' such that $T'' \sim T$. From the IH we also know that there is a T''' such that $\Gamma \vdash_a r : T'''!\,c$ and $T''' \sim T''$. We can thus pick T' to be T'''. The typing part of the conclusion follows directly from the IH, and the $\sim$ part follows by transitivity of $\sim$, as $T \sim T'' \wedge T'' \sim T''' \Rightarrow T \sim T'''$ as required. □

To allow us to prove type preservation, we define a partial order (up to $(\!|\cdot|\!)$-equivalence) for types, in Figure 6.4, $\leq$, where for two types T_1 and T_2, $T_1 \leq T_2$ if and only if all the alternatives in T_1 are contained in T_2. We can use this partial order to state a meaningful preservation property, as types will decrease during evaluation as labels get replaced and terms get specialized. For example: $A \leq$ **Case** $NeighbourhoodDensity$ **of** $\{Low \Rightarrow A; High \Rightarrow B\}$

Using this partial order, we can prove type soundness for λ_{env}. We first note how this partial order interacts with $\sim$.

Lemma 6.5. *Weak preservation of $\leq$ with respect to $\sim$: If we have two types T and T'' such that $T'' \leq T$, for all types $T' \sim T$ there is a type that is $\sim$ equivalent to T'' in order to preserve the $\leq$ ordering:*

$$T \sim T' \wedge T'' \leq T \Rightarrow \exists T_E . T_E \leq T' \wedge T'' \sim T_E$$

Proof. By noting that T' is only allowed to differ from T by adding or removing redundant alternatives by definition of $\sim$, so we can mirror these additions or removals in T to produce T_E. $T_E \leq T'$ then holds by the definition of $\leq$ as the subset relation will be preserved, and $\sim$ equivalence will not alter the codomain of the mapping. □

Recall that we defined an oracle to be well-formed if and only if all of the values for a given label have the type that Δ prescribes:

$$wf(\Theta) \overset{def}{\iff} \forall l . \forall v . v \in \Theta(l) \wedge l : t \in \Delta \Rightarrow \vdash_v v : t \wedge fl(v) = \varnothing$$

Well-formedness of oracles is obviously preserved across reductions, as we only remove values from the lists, and Δ is constant.

Lemma 6.6. *Preservation of well-formed oracle: If we evaluate a term under a well-formed oracle, the resulting oracle is also well-formed;*

$$wf(\Theta) \wedge r, \Theta \longrightarrow r', \Theta' \Rightarrow wf(\Theta')$$

Proof. Straightforward by observing the definition of reduction, as the reduction rules only ever remove values from Θ, leaving it otherwise unchanged. □

Lemma 6.7. *Preservation of free labels: If we reduce a term one step, the set of free labels will be a subset of the original:*

$$wf(\Theta) \wedge r, \Theta \longrightarrow r', \Theta' \Rightarrow fl(r') \subseteq fl(r)$$

Proof. We can intuitively see that this Lemma will hold by observing the definition of free labels and the substitutions used in the reductions. A more formal proof is by induction over the derivation of $r, \Theta \longrightarrow r', \Theta'$, and is routine. □

We can now prove type soundness by proving progress and type preservation properties as usual.

Theorem 6.3. *Progress: Given a closed term r, with no free variables or undischarged contextual dependencies (free labels, calculated by function $fl(r)$, defined in the appendix), if r is typeable, then it is either a value or is further reducible:*

$$wf(\Theta) \wedge \vdash_r r : T \downarrow \varnothing ! c \Rightarrow r \in Val \vee \exists r' \Theta', r, \Theta \longrightarrow r', \Theta'$$

Proof. We know from $\vdash r : T \downarrow \varnothing ! c$ that $\vdash r : T ! c$ and $fl(r) = \varnothing$. We proceed by induction over the derivation of $\vdash r : T ! c$. For Lam, Label and Cases, $r \in Val$.

Case Ax We know that $\Gamma = \varnothing$, so this case is impossible.

Case CTrans We can apply Kappa.

Case App Follows from the IH combined with AppArg/AppFun if either r_1 or r_2 are not values. Otherwise:

- If $r_1 = \lambda x.r_1'$, we can apply Beta.
- If $r_1 =$ **ccase** l **of** $\{\cdots\}$ or $?l$, $fl(r_1\, r_2) \neq \varnothing$, so this case is impossible.

Case LCases From the *exhaustive*$(\cdots)$ premise we know that typeable **ccase** terms are exhaustive, so $\exists i.v = vi$, and we can apply CCase.

Case Subst Follows directly from the IH. □

Type preservation can be proven directly by induction, if we reason up to $\sim$ equivalence. However, for the sake of proof modularity (and easy mapping to induction in the mechanized proof), we prove type preservation for the syntax-directed type system, and then use the proofs of soundness and completeness for this type system to get type preservation for the unrestricted system. Moreover, we can state and prove preservation under variable and label substitution independently:

Lemma 6.8. *(Weak) Type preservation under variable substitution: Substitution by terms of $\sim$-equivalent types preserves typings up to $\sim$-equivalence as long as the required compat and type constraints for the application hold. Moreover, the set of labels that will be substituted is a subset of the*

union of the sets of the original term and the term that was substituted in:

$$\Gamma, x : t \vdash_a r_1 : T_1 ! c_1 \wedge T_1 \sim T_1' \wedge T_2 \sim t \wedge \Gamma \vdash_a r_2 : T_2 ! c_2 \wedge compat(T_1', c_2) \Rightarrow$$
$$\exists c'.\exists T'.c' \subseteq (c_1 \cup c_2) \wedge T' \sim T_1' \wedge \Gamma \vdash_a [x := r_2]r_1 : T' ! c'$$

Proof. By induction on the derivation of $\Gamma \vdash_a r_1 : T_1 ! c_1$ generalizing T_1':

Case AAx $r_1 = x'$, and $T_1 = t$. Substitution will proceed differently depending on whether $x' = x$ or not.

- If $x = x'$, $[x := r_2]x' = r_2$. We know that $\Gamma \vdash_a r_2 : T_2 ! c_2$, so we can choose $c' = c_2$ and $T' = T_2$. $T' \sim T_1'$ follows by transitivity of $\sim$, since we know that $T_2 \sim t$ and $t \sim T_1$. $c_2 \subseteq (c_1 \cup c_2)$ is a trivial set identity.

- Otherwise, $[x := r_2]x' = x'$, and the result follows trivially from the assumptions, for $c'' = \emptyset$ and $T' = t$.

Case Lam $r_1 = \lambda x'.r_1'$, $T_1 = t_1 \underline{\rightarrow} T_{1,2}$ and $\Gamma, x : t, x' : t_1 \vdash r_1 : T_1 ! c_1$. By standard variable convention [Bar92] we know that $x \neq x'$ and $x \notin fv(r_2)$. We also know that $T_1' \sim t_1 \underline{\rightarrow} T_{1,2}$ from the assumptions of the Lemma. By definition of $\underline{\rightarrow}$ and $\sim$, it must be the case that there is a $T_{1,2}'$ such that $T_1' = t_1 \underline{\rightarrow} T_{1,2}'$, as $\sim$ equivalent types only differ by redundant alternatives. Since $\underline{\rightarrow}$ affects all alternatives in the same way, by prepending the same argument type, it must be the case that we can add/remove redundant alternatives in the type of $T_{1,2}$ to form $T_{1,2}'$. By the IH, choosing $T_1' = T_{1,2}'$ we have that:

$$T_{1,2} \sim T_{1,2}' \wedge compat(T_{1,2}', c') \Rightarrow T_{1,2}\exists c_1''.\exists T_1''.c_1'' \subseteq c \wedge T_1'' \sim T_{1,2} \wedge \Gamma, x' : t_1 \vdash_a [x := r_2]r_1 : T_1'' ! c'$$

The left hand side of the implication follows from the argument above and the fact that $compat(t \underline{\rightarrow} T_{1,2}', c') \Rightarrow compat(T_{1,2}', c')$, as $\underline{\rightarrow}$ does not affect the set of labels that determines a type. $T_{1,2} \sim T_{1,2}'$ follows by Lemma 6.1. We can thus pick $T' = t_1 \underline{\rightarrow} T_1''$. We can then apply ALam. The fact that $t_1 \underline{\rightarrow} T_1'' \sim t_1 \underline{\rightarrow} T_{1,2}$, follows by Lemma 6.1.

Case ALabel This case is trivial as $[x := r_2]?l = ?l$.

Case ACTrans $r_1 = l[r_1']$ and $c_1 = c_1' \cup \{l\}$. From the induction hypothesis (without rewriting T_1') we know that:

$$\exists c_1''.\exists T_1''.c_1'' \subseteq (c_1 \cup c_2) \wedge T_1'' \sim T_1' \wedge \Gamma \vdash_a [x := r_2]r_1 : T_1''!\, c_1''$$

Hence, we can pick $c' = c_1'' \cup \{l\}$, and $T' = T_1''$ and apply ACTrans. The remaining relations follow by the rest of the IH and trivial set calculations.

Case AApp $r_1 = r_{1,1} \cdot r_{1,2}$, $T_1 = T_{1,1} \cdot T_{1,2}$, with $\Gamma \vdash_a r_{1,1} : T_{1,1}'!\, c_{1,1}$ and $\Gamma \vdash_a r_{1,2} : T_{1,2}'!\, c_{1,2}$, with $T_{1,1}' \sim T_{1,1}$, $T_{1,2}' \sim T_{1,2}$, $compat(T_{1,1}, c_{1,2})$ and $compat(T_{1,2}, c_{1,1})$ The induction hypothesis, choosing for T_1' $T_{1,1}$ and $T_{1,2}$ respectively, is:

- $T_{1,1} \sim T_{1,1}' \wedge compat(T_{1,1}, c_2) \Rightarrow \exists c_1''.\exists T_1''.c_1'' \subseteq (c_{1,1} \cup c_2) \wedge T_1'' \sim T_{1,1} \wedge \Gamma \vdash_a [x := r_2]r_{1,1} : T_1''!\, c_1''$
- $T_{1,2} \sim T_{1,2}' \wedge compat(T_{1,2}, c_2) \Rightarrow \exists c_2''.\exists T_2''.c_2'' \subseteq (c_{1,2} \cup c_2) \wedge T_2'' \sim T_{1,2} \wedge \Gamma \vdash_a [x := r_2]r_{1,2} : T_2''!\, c_2''$

The assumptions on the IH follow from the Lemma assumptions and by Lemma 6.4. We can then pick $c' = c_1'' \cup c_2''$ and $T' = T_1$. We know that $T_1'' \sim T_{1,1}$ and $T_2'' \sim T_{1,2}$. We now need to prove that the compat relations are preserved. We know $compat(T_{1,1}, c_{1,2})$ and $compat(T_{1,1}, c_2)$ from the assumptions and Lemma 6.4, as before. By the definition of compat, we know $compat(T_{1,1}, c_{1,2} \cup c_2)$[1]. Given that $c_2'' \subseteq c_{1,2} \cup c_2$, by definition of compat, we have $compat(T_{1,1}, c_2'')$ as required[2]. We can use a similar argument to derive $compat(T_{1,2}, c_{1,1})$. We now can apply AApp with the chosen type and label set. The set relation follows by trivial set calculations, and the $\sim$ relation by reflexivity.

Cases ACases and ALCases $r_1 = \mathbf{ccase}\ l\ \mathbf{of}\ \{\overline{v \Rightarrow r_1}\}$, $T_1 = \mathbf{Case}\ l\ \mathbf{of}\ \{\overline{v \Rightarrow T_1}\}$, with $\overline{\Gamma \vdash r_1 : T_1!\, c_1}$ and $c_1 = \bigcup \overline{c_1}$. We can choose $\overline{T_1' = T_1}$, giving us by the IH:

$$\overline{compat(T_{1,i}, c_2) \Rightarrow \exists c_i''.\exists T_i''.c_i'' \subseteq (c_{1,i} \cup c_2) \wedge T_i'' \sim T_{1,i} \wedge \Gamma \vdash [x := r_2]r_{1,i} : T_i''!\, c_i''}^{\,i}$$

We can take $T' = \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v \Rightarrow T''}\}$ and $c' = \bigcup \overline{c''}$ and apply ACases or ALCases respectively to get the typing part of the conclusion, as all the other premises remain the same. $(\bigcup \overline{c''}) \subseteq (c_1 \cup c_2)$ follows easily by composing the subset parts of the IH and noting that $c_1 = \bigcup \overline{c_1}$. The $\sim$ part of the conclusion then follows by Lemma 6.2. □

[1]For arbitrary sets: $A \cap B = \varnothing \wedge A \cap C = \varnothing \Rightarrow A \cap (B \cup C) = \varnothing$
[2]For arbitrary sets: $A \cap B = \varnothing \wedge C \subseteq B \Rightarrow A \cap C = \varnothing$

Lemma 6.9. *Type preservation under label substitution: Substitution of labels by terms of the right type (according to Δ) will preserve typings, with a set of labels that is a subset of the original:*

$$\Gamma \vdash_a r : T\,!\,c \wedge \vdash_a v : t\,!\,\varnothing \wedge l : t \in \Delta \Rightarrow \exists c'.c' \subseteq c \wedge \Gamma \vdash_a [l := v]r : T\,!\,c'$$

Proof. This proof follows easily by induction over the typing derivation. A complete proof can be found in the appendix. □

Theorem 6.4. *(Weak) Type Preservation for the syntax-directed type system: If a term r is typeable in the syntax-directed system, and we reduce it one step, the resulting term is also typeable in the same system, with the same context and types or more specific ones, in the sense of subset or set of alternatives, respectively:*

$$\begin{aligned} &wf(\Theta) \wedge \Gamma \vdash_a r : T\,!\,c \wedge r, \Theta \longrightarrow r', \Theta' \Rightarrow \\ &\quad \exists c'T'T''.c' \subseteq c \wedge T' \leq T \wedge T'' \sim T' \wedge \Gamma \vdash_a r' : T''\,!\,c' \end{aligned}$$

Proof. By induction on the derivation of $e, \Theta \longrightarrow e', \Theta'$

Case Beta Holds by preservation under substitution, as stated in Lemma 6.8.

Case Kappa Holds by preservation under label substitution, as stated in Lemma 6.9.

Case CCase By inverting $\Gamma \vdash_a r : T\,!\,c$, we can then take $T' = T_i$. $T_i \leq T$ follows by the definition of $\leq$.

Case AppArg By inverting $\Gamma \vdash_a r : T\,!\,c$ we get:

$$\frac{\begin{array}{l}\Gamma \vdash_a r_1 : T_1\,!\,c_1 \\ \Gamma \vdash_a r_2 : T_2\,!\,c_2 \\ T' = T_1' \cdot T_2' \\ T_1' \sim T_1 \\ T_2' \sim T_2 \\ compat(T_1', c_2) \\ compat(T_2', c_1)\end{array}}{\Gamma \vdash_a r_1\, r_2 : T'\,!\,c_1 \cup c_2}\ \text{AApp}$$

From the IH we get that there is a c_3 and T_3 such that:

- $c_3 \subseteq c_2$
- $T_3 \leq T_2$
- $\Gamma \vdash_a r_2 : T_3 ! c_3$

Thus, we can take c' to be $(c_1 \cup c_3)$. We can then apply AApp if we can prove:

- $\Gamma \vdash r_1 : T_1 ! c_1$: Follows from the inversion.
- $\Gamma \vdash r_2 : T_2 ! c_2$: Follows from the IH.
- $\exists T_1'' T_2'' T_f . T_1'' \sim T_1 \wedge T_2'' \sim T_2 \wedge T_f = T_1'' \cdot T_2''$:

We know from the IH that $T_3 \leq T_2$. This means that the set of labels used to determine the type of T_3 has to be a subset of the set of labels used to determine the type of T_2. Hence, we can add just enough redundant labels to T_3 to match the labels that were removed. This means that there is a T_3', which is $\sim$-equivalent to T_3, such that $T_1 \cdot T_3'$ is defined, with a type that is smaller than the original resulting type. So, we can take T_1'' to be T_1 and T_2'' to be T_3'.

- $compat(T_1, c_3)$: Follows from $c_3 \subseteq c_2$ and the definition of compat.
- $compat(T_3', c_1)$: This holds as T_3' uses the same set of labels to determine its type as T_2, and we know $compat(T_2, c_1)$ from the IH.

Case AppFun Holds by a similar argument as case AppArg □

Theorem 6.5. *(Weak) Type Preservation: If a term r is typeable, and we reduce it one step, the resulting term is also typeable, with the same context and type or more specific ones, in the sense of subset or set of alternatives, respectively. Moreover, the set of free labels will also be preserved:*

$$wf(\Theta) \wedge \Gamma \vdash_r r : T ! c \wedge r, \Theta \longrightarrow r', \Theta' \Rightarrow$$
$$\exists c' T' . c' \subseteq c \wedge T' \leq T \wedge \Gamma \vdash_r r' : T' ! c''$$

Proof. Straightforward by combining soundness and completeness of $\vdash_a$ with type preservation for $\vdash_a$ and the auxiliary Lemmas for the interaction between $\leq$ and $\sim$: We can apply Theorem 6.2 (Completeness of $\vdash_a$) to the typing judgement to get:

$$\exists T'' . \Gamma \vdash_a e : T'' ! c \wedge T \sim T''$$

We can then apply Theorem 6.4 (Type preservation for $\vdash_a$) to get:

$$\exists c'T'.c' \subseteq c \wedge T' \leq T'' \wedge \Gamma \vdash_a e' : T' ! c' \wedge T \sim T''$$

Applying Theorem 6.1 (Soundness of $\vdash_a$) to the typing judgement then yields:

$$\exists c'T'.c' \subseteq c \wedge T' \leq T'' \wedge \Gamma \vdash e' : T' ! c' \wedge T \sim T''$$

By Lemma 6.5, we know that: $\exists T_f.T_f \sim T'' \wedge T' \leq T_f$. By transitivity of $\sim$, $T \sim T_f$, so we can finally apply Subst to get:

$$\exists c'T'.c' \subseteq c \wedge T' \leq T \wedge \Gamma \vdash e' : T' ! c'$$

□

6.3 Type inference for λ_{env}

We will now present a type inference system. The main challenge is the non-determinism present in the AApp rule. In order to design a type inference algorithm for $\vdash_a$ we now need to be able to, when inferring a type for an application, compute two types T_1' and T_2' that are $\sim$-equivalent to the types of the argument and function respectively, such that $T_1 \cdot T_2$ is defined.

We achieve this by making sure the argument contains just as many redundant clauses as needed for application. We do so by calculating a (possibly redundant) $\sim$-equivalent type for the argument, that will succeed with the classical application rule. We can use the $\sqsubseteq$ relation to compute a canonical type for any type T as the least element T' according to this relation, such that $T' \sqsubseteq T$. We prove that this is decidable by giving a definition of an algorithm for computing it:

Lemma 6.10. *Decidability of canonical type: Calculating T, such that for some T', $T = canonical(T')$, is decidable.*

Proof. A canonical type T for a type T' was defined as a type that is equivalent to T', but has no redundant labels. In order to define formally what it means for a label to be redundant we define partial lookups on representations:

$$(\!|T|\!) \downarrow lm' = \{lm \setminus lm' \mapsto T \mid lm \mapsto T \in (\!|T|\!), lm' \subseteq lm\}$$

$\boxed{T \cdot_{\mathscr{U}} T = \langle S, T \rangle}$

$$T \cdot_{\mathscr{U}} T' = compute\_app_{\mathscr{U}}(canonical(T), \varnothing, inflate_T(canonical(T), canonical(T')))$$

$$inflate(T, (lm, t)) = \{lm \cup lm' \mapsto t \mid lm' \mapsto t \in (\!|T|\!), lm \subseteq lm'\}$$

$$inflate^*(T, T') = \{m \mid lm \to t' \in (\!|T'|\!), m \in inflate(T, (lm, t'))\}$$

$$T = inflate_T(T_1, T_2) \overset{def}{\iff} (\!|T|\!) = inflate^*(T_1, T_2)$$

$\boxed{compute\_app_{\mathscr{U}}(T, lm, T') = \langle S, T''' \rangle}$

$$\frac{\begin{array}{l}\textbf{fresh}\ X \\ t = (\!|T|\!)(lm) \\ S = \mathscr{U}(t \to X, t')\end{array}}{compute\_app_{\mathscr{U}}(t', lm, T) = \langle S, S(X) \rangle}\ \textsc{CauSimpTy}$$

$$\frac{\overline{compute\_app_{\mathscr{U}}((S_{i-1} \circ ... \circ S_1)(T_i), lm \cup \{(l, v_i)\}, (S_{i-1} \circ ... \circ S_1)(T)) = \langle S_i, T_i' \rangle}^{\,i}}{compute\_app_{\mathscr{U}}(\textbf{Case}\ l\ \textbf{of}\ \{\overline{v_i \Rightarrow T_i}^{\,i}\}, lm, T) = \langle (S_n \circ ... \circ S_1), \textbf{Case}\ l\ \textbf{of}\ \{\overline{v_i \Rightarrow (S_n \circ ... \circ S_1)(T_i')}^{\,i}\} \rangle}\ \textsc{CauAlt}$$

(CauAlt)

Figure 6.6: Auxiliary definitions for type inference.

We then say that a label is redundant, if and only if for all the values it can have, the resulting representation from partially projecting all the label-value mappings for that label is the same:

$$redundant(l, T) \overset{def}{\iff} \exists rep. \forall v.\ l : t \in \Delta \wedge \vdash_v v : t! \varnothing \Rightarrow (\!|T|\!) \downarrow \{(l, v)\} = rep$$

Computing a canonical type representation is then just a matter of removing all the mappings for redundant labels:

$$(\!|canonical(T)|\!) = \{\{(l, v) \mid (l, v) \in lm, \neg redundant(l, T)\} \mapsto t \mid lm \mapsto t \in (\!|T|\!)\}$$

There are several ways to convert a representation into a type, but they are all $(\!|\cdot|\!)$-equivalent, by the definition of representation. Since *canonical* was defined in terms of $\sqsubseteq$, which also works on representations, its definition is also only up to $(\!|\cdot|\!)$-equivalence (it does not define a function in the set-theoretical sense). □

We then inflate the canonical type by adding the necessary (redundant) label-value pairs for every alternative. We achieve this by using the function *inflate* defined in Figure 6.7. For

example, if we revisit the previous example for expanding types:

$$\begin{aligned} e_1 : & \textbf{ Case } NeighbourhoodDensity \textbf{ of} \\ & \quad \{Low \Rightarrow A \to B; High \Rightarrow A \to C\} \\ e_2 : & A \end{aligned}$$

In this case we would need to inflate the type of e_2 so that it covers all the cases in the type of e_1. We then have:

$$\begin{aligned} & inflate(\textbf{ Case } NeighbourhoodDensity \textbf{ of} \\ & \quad \{Low \Rightarrow A \to B; High \Rightarrow A \to C\}, A) = \\ & \textbf{Case } NeighbourhoodDensity \textbf{ of } \{Low \Rightarrow A; High \Rightarrow A\} \end{aligned}$$

Note that there are several ways to revert a representation into a type. However, because of the way we compute the result type of an application, order/nesting is irrelevant so all of these types are equivalent.

After inflating the type we are then, up to instantiation of variables, in a situation where the regular App rule would be able to type the term, so we can just perform unification as usual (denoted by $\mathscr{U}$). As usual, when inferring a type for an application term we compose substitutions. Similarly, when inferring types for **ccase** terms we perform the composition of the substitutions gradually for each alternative. This is reflected in the $compute\_app_{\mathscr{U}}$ definition. The soundness of $compute\_app_{\mathscr{U}}$ is defined with respect to $compute\_app$:

Lemma 6.11. *Soundness of $compute\_app_{\mathscr{U}}$: If $compute\_app_{\mathscr{U}}$ succeeds, the result is a type and a substitution, such that if we apply the substitution to both argument and result types, compute_app would succeed with the same result:*

$$\langle T, S \rangle = compute\_app_{\mathscr{U}}(T_1, lm, T_2) \Rightarrow T = compute\_app(S(T_1), lm, S(T_2))$$

Proof. By induction over T_1.

Case $T_1 = t_1$ $T = S(X)$, $S = \mathscr{U}(t_2 \to X, t_1)$ and $t_2 = (\!|T_2|\!)(lm)$, with X fresh. By definition of unification we know that $S(t_1) = S(t_2 \to X) = S(t_2) \to S(X)$. We can thus rewrite the conclusion to $S(X) = compute\_app(S(t_2) \to S(X), lm, S(T_2))$. Since substitutions only affect types, $(\!|S(T_2)|\!)(lm) = S((\!|T_2|\!)(lm)) = S(t_2)$, so we can apply CASimpTy to get the conclusion.

Case T_1 = **Case** l **of** {...} By the IH we get:

$$\overline{T'_i = compute\_app(S_i \ldots S_1(T_{1,i}), lm, S_i \ldots S_1(T_2))}$$

By considering the effect of the substitutions in the result of *compute_app*, we know that there is are $\overline{T''}$ such that $\overline{T'} = S_i \ldots S_1\overline{T''}$, so we can rewrite the IH as:

$$\overline{S_i \ldots S_1(T''_i) = compute\_app(S_i \ldots S_1(T_{1,i}), lm, S_i \ldots S_1(T_2))}$$

By applying $S_n \circ \ldots \circ S_{i+1}$, on both sides of the equality we get:

$$\overline{S_n \ldots S_1(T''_i) = compute\_app(S_n \ldots S_1(T_{1,i}), lm, S_n \ldots S_1(T_2))}$$

We can then apply CAAlt to get the conclusion. □

This result induces a soundness result for $\cdot_{\mathscr{U}}$ trivially, by the definition of $\cdot_{\mathscr{U}}$:

Corollary 6.1. *Soundness of* $\cdot_{\mathscr{U}}$*:*

$$\langle S, T\rangle = T_1 \cdot_{\mathscr{U}} T_2 \Rightarrow T = S(canonical(T_1)) \cdot S(inflate(canonical(T_1), canonical(T_2)))$$

Moreover, we state a standard property for preservation of typings under type variable substitutions:

Lemma 6.12. *Preservation of typings under type variable substitutions:*

$$\Gamma \vdash_a r : T\,!\,c \Rightarrow S\Gamma \vdash_a r : ST\,!\,c$$

Proof. The proof is standard, and proceeds by induction over the structure of typing derivations. □

We present the algorithm through inference rules, $inf(\Gamma, e)$ in Figure 6.7. We can see that the "relation" defined is a partial function, as its definition is structural on e, and we don't introduce new variables in the premises that cannot be determined from the information we are given. We assume that the algorithm used for computing a fresh type variable for a given environment is deterministic. Most rules are standard extensions of the Hindley-Milner type inference algorithm. The ICCases rule uses a similar logic as the ICApp rule in incrementally building substitutions, by applying the composed substitution obtained so far

to the environment in each inference step. This ensures that the substitutions obtained are compatible with each other, as will be seen more concretely in the proof of soundness.

With this notion of type equivalence, we can now prove soundness of type inference:

Theorem 6.6. *Restricted soundness of type inference: If the inference algorithm succeeds for expression e, then its result is a substitution for the environment, a type and a context set, such that they constitute a valid typing for e in the syntax-directed system:*

$$inf(\Gamma, e) = \langle S, T, c\rangle \Rightarrow S\Gamma \vdash_a e : T \,!\, c$$

Proof. By induction over the derivation of $inf(\Gamma, e) = \langle S, T, c\rangle$

Case IAx We can apply AAx.

Case ILam $e = \lambda x.e'$ and $T = S(X)\underline{\rightarrow}T'$. By the IH and the definition of substitution we have:

$$S\Gamma, x : S(X) \vdash_a e' : T ! c$$

We can then apply ALam to get:

$$S\Gamma \vdash \lambda x.e' : S(X)\underline{\rightarrow}T' ! c$$

Case IApp By the IH we have:

- $S_1\Gamma \vdash_a r_1 : T_1 ! c_1$
- $S_2S_1\Gamma \vdash_a r_2 : T_2 ! c_2$

We can apply Lemma 6.12 repeatedly to the previous to get:

- $SS_2S_1\Gamma \vdash_a r_1 : SS_2T_1 ! c_1$
- $SS_2S_1\Gamma \vdash_a r_2 : ST_2 ! c_2$

We can then apply AApp to the previous if we can prove:

$$\exists T_1'T_2'T'.T' = T_1' \cdot T_2' \wedge T_1' \sim SS_2T_1 \wedge T_2' \sim ST_2$$

$\boxed{inf(\Gamma, r) = \langle S, T, c\rangle}$

$$\frac{x : t \in \Gamma}{inf(\Gamma, x) = \langle \mathbf{id}, t, \varnothing\rangle}\ \textsc{ICCAx}$$

$$\frac{\begin{array}{l} inf(\Gamma, x : X, r) = \langle S, T, c\rangle \\ \mathbf{fresh}\ X \end{array}}{inf(\Gamma, \lambda x.r) = \langle S, S(X) \underline{\rightarrow} T, c\rangle}\ \textsc{ICCLam}$$

$$\frac{\begin{array}{l} inf(\Gamma, r_1) = \langle S_1, T_1, c_1\rangle \\ inf(S_1(\Gamma), r_2) = \langle S_2, T_2, c_2\rangle \\ \langle T', S\rangle = S_2(T_1) \cdot_{\mathscr{U}} T_2 \\ compat(S(S_2(T_1)), c_2) \\ compat(S(T_2), c_1) \\ S' = S \circ S_2 \circ S_1 \end{array}}{inf(\Gamma, r_1\, r_2) = \langle S', T', c_1 \cup c_2\rangle}\ \textsc{ICCApp}$$

$$\frac{l : t \in \Delta}{inf(\Gamma, ?l) = \langle \mathbf{id}, t, \varnothing\rangle}\ \textsc{ICCLabel}$$

$$\frac{\begin{array}{l} inf(\Gamma, r) = \langle S, T, c\rangle \\ l : t \in \Delta \end{array}}{inf(\Gamma, l[r]) = \langle S, T, c \cup \{l\}\rangle}\ \textsc{ICCCTrans}$$

$$\frac{\begin{array}{l} \dfrac{l : t \in \Delta}{\vdash_v v_i : t}{}^{i} \\ \dfrac{}{inf((S_{i-1} \circ .. \circ S_1)(\Gamma), r_i) = \langle S_i, T_i, c_i\rangle}{}^{i} \\ c' = \cup \overline{c_i}^{\,i} \\ S' = S_n \circ ... \circ S_1 \\ exhaustive(\overline{v}, t) \end{array}}{inf(\Gamma, \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow r_i}^{\,i}\}) = \langle S', \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow (S_n \circ ... \circ S_{i+1})(T_i)}^{\,i}\}, c'\rangle}\ \textsc{ICCCases}$$

$$\frac{inf(\Gamma, r) = \langle S, T, c\rangle}{inf(\Gamma, r) = \langle S, T, c\rangle}\ \textsc{ICCPop}$$

Figure 6.7: Type inference algorithm.

We can choose $T_1' = S(canonical(S_2T_1))$, and $T_2' = S(inflate(canonical(S_2T_1), canonical(T_2)))$. The $\sim$ parts of the conclusion then follow by $\sim$-equivalence of types after *canonical* and *inflate* (easy to see from the definition of *canonical* and *inflate*). *compat* is preserved over *canonical* and *inflate*, as it is calculated on the *canonical* type, by definition. The fact that $S(canonical(S_2T_1)) \cdot S(inflate(canonical(S_2T_1), canonical(T_2)))$ is defined and equal to T' then follows by Corollary 6.1 (Soundness of $\cdot_{\mathscr{U}}$).

Case ICases By the IH we have:

$$\overline{(S_i \circ ... \circ S_1)(\Gamma) \vdash_a r_i : T_i ! c_i}$$

We can apply Lemma 6.12 with the substitution $(S_n \circ ... \circ S_{i+1})$, yielding:

$$\overline{(S_n \circ ... \circ S_1)(\Gamma) \vdash_a r_i : (S_n \circ ... \circ S_{i+1})(T_i) ! c_i}$$

We can then apply ACases to get the desired result as the other premises are also premises of ICases. □

Theorem 6.7. *Soundness of type inference: If the inference algorithm succeeds for expression e, then its result is a substitution for the environment, a type and a context set, such that they constitute a valid typing for e:*

$$inf(\Gamma, e) = \langle S, T, c \rangle \Rightarrow S\Gamma \vdash e : T \; ! \; c$$

Proof. Follows trivially by combining the previous restricted soundness Theorem with Theorem 6.1 (Soundness of $\vdash_a$). □

Conjecture 6.1. *Restricted completeness of type inference: If e has type T under* Γ*, we can infer a triple* $\langle S_I, T_I, c_I \rangle$ *such that:*

$$S_A\Gamma \vdash_{ai} e : T \; ! \; c \Rightarrow \exists S_U . inf(\Gamma, e) = \langle S_I, T_I, c_I \rangle \wedge S_U T_I = T \wedge S_U S_I \Gamma = S_A \Gamma$$

We believe that this property holds and justify it by following the argument in the beginning of the section. Specifically, we argued that the only source of non-determinism that cannot be resolved by unification is the $\sim$-equivalences in the AApp rule. We conjecture that this non-determinism can be resolved, and the algorithm will always succeed and return a valid result. The property as stated above is a generalisation of completeness, which we believe will be needed in order to prove completeness. This generalisation is standard in proofs for Hindley-Milner style systems and makes completeness amenable for structural induction. A full formal proof is left for future work.

By joining Theorem 6.2 and Lemma 6.1 we get our desired completeness property for the unrestricted system:

Conjecture 6.2. *Completeness of type inference: If e has type T under* Γ*, we can infer a triple* $\langle S, T', c' \rangle$ *such that:*

$$\Gamma \vdash e : T \mid c \Rightarrow \exists S_U . inf(\Gamma, e) = \langle S_I, T_I, c_I \rangle \wedge S_U T_I \sim T \wedge S_U S_I \Gamma = \Gamma$$

6.4 Polymorphism

We support Hindley-Milner style polymorphism [DM82], by allowing universally quantified variables at the top level. We then allow instantiation of variables and generalisation of types. This is embodied in the rules Inst and Gen. Instantiation of types is reflected in the partial order $\preceq$, which is defined as usual as:

$$\forall \overline{X}.\, T \preceq \forall \overline{X'}.T' \stackrel{def}{\iff} T' = [\overline{T/X}]T \wedge \overline{X'_i} \cap ftv(\forall \overline{X}.T) = \varnothing$$

Generalisation is allowed as long as the type is not bound in the environment, as would be expected. This polymorphism is limited in that it does not allow generalisation to happen only within alternatives. Ways to extend this mechanism to be more general are discussed in Section 8.3.

We can also extend the type inference algorithm to take into account polymorphism, by instantiating and generalising in the appropriate places, as usual:

Given the stratification of the types, and the fact that only (potentially polymorphic) simple types are allowed in environments, we need to state well-formedness of variable environments separately:

Definition 6.1. *Well-formedness of environments: An environment* Γ *is said to be well-formed if all of the types it maps variables to are of the form* $\forall X_1. \ldots . \forall X_n.t$*:*

$$wf_\Gamma(\Gamma) \stackrel{def}{\iff} x : \sigma \in \Gamma \Rightarrow \exists t.\sigma \preceq t$$

The revised system is still type-sound, and the inference algorithm sound and (probably) complete:

Theorem 6.8. *Progress: Given a closed term e, with no free variables or undischarged contextual dependencies (free labels, calculated by function* $fl(e)$*, defined in the appendix), if e is typeable, then*

$\boxed{\Gamma \vdash_P e : \sigma ! c}$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_P x : \sigma ! \varnothing} \ \textsc{PAx}$$

$$\frac{\Gamma, x : t \vdash_P e : T ! c}{\Gamma \vdash_P \lambda x.e : t \underline{\rightarrow} T ! c} \ \textsc{PLam}$$

$$\frac{l : t \in \Delta}{\Gamma \vdash_P ?l : t ! \varnothing} \ \textsc{PLabel}$$

$$\frac{\begin{array}{l}\Gamma \vdash_P e : T ! c \\ l : t \in \Delta\end{array}}{\Gamma \vdash_P l[e] : T ! c \cup \{l\}} \ \textsc{PCTrans}$$

$$\frac{\begin{array}{l}\Gamma \vdash_P e_1 : T_1 ! c_1 \\ \Gamma \vdash_P e_2 : T_2 ! c_2 \\ T' = T_1 \cdot T_2 \\ compat(T_1, c_2) \\ compat(T_2, c_1)\end{array}}{\Gamma \vdash_P e_1\, e_2 : T' ! c_1 \cup c_2} \ \textsc{PApp}$$

$$\frac{\begin{array}{l}l : t \in \Delta \\ \overline{\vdash_v v_i : t}^{\,i} \\ \overline{\Gamma \vdash_P e_i : T_i ! c_i}^{\,i} \\ c' = \cup \overline{c_i}^{\,i} \\ exhaustive(\overline{v}, t)\end{array}}{\Gamma \vdash_P \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow e_i}^{\,i}\} : \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i}^{\,i}\} ! c'} \ \textsc{PCases}$$

$$\frac{\begin{array}{l}\Gamma \vdash_P e : T ! c \\ T \sim T'\end{array}}{\Gamma \vdash_P e : T' ! c} \ \textsc{PSubst}$$

$$\frac{\begin{array}{l}\Gamma \vdash_P e : \sigma' ! c \\ \sigma' \preccurlyeq \sigma\end{array}}{\Gamma \vdash_P e : \sigma ! c} \ \textsc{PInst}$$

$$\frac{\begin{array}{l}\Gamma \vdash_P e : \sigma ! c \\ X \notin fv(\Gamma)\end{array}}{\Gamma \vdash_P e : \forall X.\sigma ! c} \ \textsc{PGen}$$

Figure 6.8: Type system for the polymorphic system.

$\boxed{inf(\Gamma, r) = \langle S, T, c\rangle}$

$$\frac{x : \sigma \in \Gamma \quad \langle t, F'\rangle = inst(F, \sigma)}{inf(\Gamma, x) = \langle \mathbf{id}, t, \varnothing\rangle}\ \textsc{IAx}$$

$$\frac{inf(\Gamma, x : X, r) = \langle S, T, c\rangle \quad \mathbf{fresh}\ X}{inf(\Gamma, \lambda x.r) = \langle S, S(X) \underline{\rightarrow} T, c\rangle}\ \textsc{ILam}$$

$$\frac{\begin{array}{l} inf(\Gamma, r_1) = \langle S_1, T_1, c_1\rangle \\ inf(S_1(\Gamma), r_2) = \langle S_2, T_2, c_2\rangle \\ \langle T', S\rangle = S_2(T_1) \cdot_{\mathscr{U}} T_2 \\ compat(S(S_2(T_1)), c_2) \\ compat(S(T_2), c_1) \\ S' = S \circ S_2 \circ S_1 \end{array}}{inf(\Gamma, r_1\, r_2) = \langle S', T', c_1 \cup c_2\rangle}\ \textsc{IApp}$$

$$\frac{l : t \in \Delta}{inf(\Gamma, ?l) = \langle \mathbf{id}, t, \varnothing\rangle}\ \textsc{ILabel}$$

$$\frac{inf(\Gamma, r) = \langle S, T, c\rangle \quad l : t \in \Delta}{inf(\Gamma, l[r]) = \langle S, T, c \cup \{l\}\rangle}\ \textsc{ICTrans}$$

$$\frac{\begin{array}{l} l : t \in \Delta \\ \overline{\vdash_v v_i : t}^{\,i} \\ \overline{inf((S_{i-1} \circ .. \circ S_1)(\Gamma), r_i) = \langle S_i, T_i, c_i\rangle}^{\,i} \\ c' = \cup\, \overline{c_i}^{\,i} \\ S' = S_n \circ ... \circ S_1 \\ exhaustive(\overline{v}, t) \end{array}}{inf(\Gamma, \mathbf{ccase}\ l\ \mathbf{of}\ \{\overline{v_i \Rightarrow r_i}^{\,i}\}) = \langle S', \mathbf{Case}\ l\ \mathbf{of}\ \{\overline{v_i \Rightarrow (S_n \circ ... \circ S_{i+1})(T_i)}^{\,i}\}, c'\rangle}\ \textsc{ICases}$$

$$inst(\forall \overline{X}.T) = [\overline{X := X'}]\, T, \mathbf{fresh}\ \overline{X'}$$

Figure 6.9: Inference algorithm for the polymorphic system.

it is either a value or is further reducible:

$$wf(\Theta) \wedge fl(e) = \varnothing \wedge \vdash_P e : \sigma \,!\, c \Rightarrow e \in Val \vee \exists e'\Theta', e, \Theta \longrightarrow e', \Theta'$$

Theorem 6.9. *Type Preservation: If a term e is typeable, and we reduce it one step, the resulting term is also typeable, with the same context and type or more specific ones, in the sense of subset or set of alternatives, respectively. Moreover, the set of free labels will also be preserved:*

$$wf_\Gamma(\Gamma) \wedge wf(\Theta) \wedge \Gamma \vdash_P e : \sigma \,!\, c \wedge e, \Theta \longrightarrow e', \Theta' \Rightarrow$$
$$\exists c'\sigma'.c' \subseteq c \wedge \sigma' \leq \sigma \wedge \Gamma \vdash_P e' : \sigma' \,!\, c''$$

Theorem 6.10. *Soundness of type inference: If the inference algorithm succeeds for expression e, then its result is a substitution for the environment, a type and a context set, such that they constitute a valid typing for e:*

$$inf(\Gamma, e) = \langle S, T, c \rangle \Rightarrow S\Gamma \vdash_P e : T \,!\, c$$

Conjecture 6.3. *Completeness of type inference: If e has type T under* Γ*, we can infer a triple* $\langle S, T', c' \rangle$ *such that:*

$$wf_\Gamma(\Gamma) \wedge \Gamma \vdash_P e : T \,!\, c \Rightarrow \exists T'S'.inf(\Gamma, e) = \langle S, T', c' \rangle \wedge S'T' \sim T \wedge S'S\Gamma = \Gamma$$

Proofs are identical to the ones for the monomorphic calculus, with the exception of reasoning steps with substitutions and unification that are standard for polymorphic lambda calculus inference proofs. They can be found in full in Appendix B.

6.5 Discussion

In this section we analysed the problem of adding context-dependent types to a functional programming language. In order to do so, we analyzed the $\lambda_{\downarrow[]}$ system that we proposed in chapter 4. This calculus provides contextual information through an oracle and allows us to examine the metatheory of a context-dependent system. We then defined a construct for context-dependent types, **ccase** , which allows us to introduce a restricted conditional of dependent types. The type system that we developed allows us to keep the same semantics we outlined for context-dependent values, in the presence of context dependent types. In particular, we developed rules that specify in which cases application terms comprising two context-dependent values are valid. Then, aligned with our goal of not mandating any artificial ordering of alternatives, we specify an equivalence relation, deeming types equivalent if they depend on context in an equivalent away. We also defined an inference algorithm, that

is sound and complete, allowing us to liberate the programmer from having to determine the context-dependent types themselves.

This allows us to retain the properties that we outlined in our design goals, and yet augment it with context-dependent types. The example that we provided shows how this system can be used to naturally express an application where context-dependent types are a useful abstraction. In the next section, we will attempt to regain the constructs for knowledge representation we presented in Chapter 5, by adding them to the calculus and forming a practical high-level programming language.

CHAPTER 7

A High Level Language with Context-Dependent Types

In Chapter 5 we presented an exploration of the design space for context-aware values in a strongly typed functional setting. We explored composability and context provision for context-aware computations in the setting of a DSL in Haskell. We also evaluated how it could be used to make typical context-aware applications more concise and safer. Then, in chapter 6 we explored a restriction of dependent types whereby we allow for types to only depend on context, modally, whilst retaining composability. In order to simplify the metatheoretical analysis of type soundness and the properties of inference, that calculus was very basic. In this Chapter we will discuss how to extend the previous calculus to include higher-level features present in practical programming languages, as well as some of the context provision mechanisms that were discussed in Chapter 5. We will then use this high-level language to present a practical case study of the application of these ideas to a real-world context-aware problem, which is outside the scope of current context-aware programming approaches. Indeed, current approaches will not provide any type-checking in the presence of context-dependent types. We then compare our solution with a current state of the art practical functional programming language, Haskell, as it allows us to provide some of the same safety guarantees, although at the cost of clarity.

In this Chapter we present:

- An informal discussion of how to extend the context-dependent types calculus in order to include practical programming features such as data types and recursion
- A discussion of how to include the missing features from the context-dependent values EDSL

- A case study for evaluation of the approach, comparing our language and Haskell in the implementation of a context-aware application that is outside the scope of current context-awareness frameworks and languages.

7.1 Extensions

In this section we will discuss how to extend the calculus presented in the previous chapter to turn it into a practical programming language.

7.1.1 Data Types, Recursion and Side Effects

The calculus that we presented in the previous chapter, although interesting, is missing certain features, such as data types and recursion, which are essential in a practical functional programming language. These extensions are well-known as extensions for Hindley-Milner type systems such as ML [MTM97] or Haskell [M$^+$10]. For instance we can add top-level (monomorphic) algebraic datatype declarations with Haskell-style syntax, of the form:

$$data\ \tau = \kappa_1\, t_{1,1} \ldots t_{1,a(1)} \mid \ldots \mid \kappa_n\, t_{n,1} \ldots t_{n,a(n)}$$

Where we are defining a data type T, formed of n constructors κ_i, each of arity $a(i)$. We also allow constructors to appear at the expression level, to form constructor application expressions. The types of these constructors are then given by:

$$\vdash \kappa_i : t_{i,1} \to \ldots \to t_{i,a(i)} \to \tau ! \varnothing$$

We also add case expressions of the form:

$$\textbf{case}\ e\ \textbf{of}\ \{\overline{\kappa_i\, x_1 \ldots x_{a(i)} \Rightarrow e_i}^{\,i}\}$$

Allowing us to branch depending on the constructor used to build a data type, and access the data contained within. These are a simplified version of the ones present in Haskell, as they do not allow arbitrary pattern matching, just destruction of the data type. Typing and reduction of case expressions of this form is standard.

Recursion can also be added as we discussed in section 2.1.2, by adding a **fix** operator, and

appropriate typing and reduction rules. We can then support a top-level recursive assignment statement by translating recursion into fixpoints as standard.

Moreover, we add support for side effects, through an IO monad, as also done in Haskell [Wad93]. This involves extending the types with the form $IO(t)$, and providing a primitive $bind : \forall X.\forall Y.IO(X) \rightarrow (X \rightarrow IO(Y)) \rightarrow IO(Y)$. We can then write statements of the form $x \leftarrow e$, which are desugared to $bind\ e\ (\lambda x.$. These hanging lambdas are scoped by using a keyword **do** , which determines the end of the scope of all the lambdas. Naked expressions are allowed within do blocks, and are also translated into $bind e(\lambda x.$, with x fresh for the block. The last expression in a do block must then be an expression of type $IO(t)$ (considering all the assumptions in the environment from the binds), which will be the final type of the block.

We argue here that these extensions are fairly conservative and should not affect the properties that we have proven. Note in particular how we do not allow for alternative types (T) to appear as arguments to constructors. For some of the simpler data type examples, such as simple sums and products, lists and trees, this argument can be done by presenting the Church encodings [Bar92] of these data types. However, since these encodings in general require higher rank polymorphism, we cannot present a general result using this technique. We leave a full formalisation of the high-level language for future work, where we will extend the typing and semantics for each of these extensions and show that none of the properties are affected.

7.1.2 Context Provision

In order to show one way in which the label system can accommodate more complex ontologies, we will also encode the individual/feature mechanism used in Chapter 5. We add two top-level statements **individual** i and **feature** $f : t$, analogous to the ones presented previously. We then define Δ as the product of the set of individuals and feature typings. In general, in order to support a knowledge representation structure, we need a way to define a set of all types of contextual information that can be accessible, and a way of computing Δ in a sound way.

7.1.3 Context definition

We have so far assumed that all the relevant contextual information has been categorised in labels, and we have not mentioned the processing that must happen to produce valid contextual information from raw sensor readings. In this section we will informally argue that this assumption is reasonable and present some language level features that could help with

this. We can start thinking about the problem from the sensor level, whereby we have a set, say Δ_s, which contains all the raw sensor values that we can use. The problem of processing this information is then that of producing context-dependent context producing functions. For instance, we can threshold raw density readings into two tiers, high density or low density, potentially after performing some processing to remove noise and avoid excessive state flapping. Thus, we can provide the high-level language with the capacity of defining new contextual values (at the top-level), for instance with the syntax **context** $l := e$. For instance, to perform the aforementioned thresholding (and assuming standard numeric comparison and if-then-else constructs):

```
context DensityTier :=
  if ?Density > 5 then HighDensity else LowDensity
```

These definitions can refer to other custom contexts as long as no cycles are produced. Thus, we can straightforwardly order the set of context definitions so that every statement only refers to ones that precede it. This also allows us to incrementally build a set of labels Δ, and perform type inference on the right hand side of context definition statements. Thus, the compiler would traverse the program looking for these definitions, order them, and collect the complete set of contextual labels Δ that are available for the application developer.

This feature also mitigates the limitation that we do not allow for arbitrary expressions to be scrutinees of **ccase** expressions. We think that this strict separation of context definition and usage is not too restrictive as usually the set of contextual values that can be used as cross-cutting concepts throughout the program can be defined beforehand.

7.2 CDT: A high-level language with context-dependent types

Based on the extensions described above, the high-level language we will use in our evaluation has the following grammar:

$$
\begin{aligned}
&stmt \in Stmt ::= e \mid x \leftarrow e \\
&e \in Expr ::= x \mid \lambda x.e' \mid e\,e' \mid ?l \mid \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow e_i}^{\,i}\} \mid l[e] \mid \kappa \mid \mathbf{case}\, e\, \mathbf{of}\, \{\overline{\kappa \bar{x} \Rightarrow e}\} \mid \mathbf{do}\, \overline{stmt} \\
&tl \in TopLevel ::= \mathbf{data}\, \tau\, \overline{X} = \kappa_1\, \bar{t}_1 \mid \ldots \mid \kappa_n\, \bar{t}_n \mid \mathbf{context}\, l = e \mid x : t \mid x = e \mid \mathbf{individual}\, i \mid \mathbf{feature}\, f : t \\
&P \in Program ::= \overline{tl} \\
&l \in Label ::= Individual \triangleright Feature \mid Feature
\end{aligned}
$$

Figure 7.1: Grammar of CDT

We have implemented a parser, interpreter and type inference algorithm for a subset of CDT

in Haskell, following Mark Jones' implementation [Jon99] for the Haskell-like extensions. We used this implementation to check the well-formedness of the examples we will present in the evaluation.

We should highlight at this point that even though CDT looks similar to the DSL from Chapter 5 it is based on completely independent semantics, and does not compile to Haskell. However, it uses Haskell-style syntax and some of the same syntax from the Haskell DSL, as it reflects similar functionality. For the sake of reference, the first example from Chapter 5 would be the following in CDT (assuming the definition of the Haskell standard library functions in CDT):

```
individual User
feature IsLocatedAt :: Location

data Shop = Shop String Location
slocation s = case s of { Shop n l ⇒ l }

allShops = ...

nearestShops = sortBy (λs1. λs2.
  compare (distance ?IsLocatedAt (slocation s1))
          (distance ?IsLocatedAt (slocation s2))) allShops

main = loop IsLocatedAt[print (take 10 nearestShops)]
```

Notice how the knowledge base is managed by the operational semantics of the language, instead of explicitly.

7.3 Case study: Telephony

Even though so far we have presented examples from the ubiquitous computing domain, the ideas in this thesis are in fact generally applicable to any situations where we have different behaviour depending on some globally relevant value. We can thus use our system to provide safety while maintaining a natural programming style when expressing these problems. In this section we present one such example, which comes from the implementation of telephony systems and contrast current approaches for achieving the desired behaviour in functional programming with our approach. We choose Haskell as the language to compare with as it has a strong emphasis on purity and static types to enforce properties of programs.

One area where we currently observe a clash between the requirements and the abstraction techniques that are being used is in telephony systems. Currently, we commonly have situations where networks using different calling protocols need to interact with each other. For instance, enterprise protocols typically support high-cost features such as placing a caller on hold or transferring calls, which may be infeasible in public networks. This implies that the switches within the network need to implement all of these protocols and be able to switch between them as required. To illustrate this and how our system would help with the problem, we will consider a protocol P_1, which allows the users to transfer calls instead of accepting them, and another protocol P_2 that doesn't. In this circumstance, in order to have a total implementation of the call handling routine, the developer has to implement the transfer and the accept case in P_2, but only the accept case in P_1. Our goal is to have the compiler know enough about the application through the structure of the types to be able to enforce this at compile-time.

There are two common ways of implementing this functionality: using a common supertype for the return values or using tagged unions. In order to encode this in Haskell we can resort to algebraic data types, encoding the data of the domain under consideration. On a first approach we will consider a straightforward encoding whereby we enumerate the protocols and types of handles that we can use:

```
type Handle  = Int
type THandle = Int
type Status1 = Int
type Status2 = Int

data AcceptedOrTransfer =
  Accept Handle | Transfer THandle
data Protocol = P1 | P2
data ProtoStatus = S1 Status1 | S2 Status2
```

The type *AcceptedOrTransfer* is a common supertype for the handles that we want to return. The main program consists of querying the protocol that is being used (from the environment) and then executing 'call' repeatedly (caching that environment, for consistency reasons). The 'call' routine, then calls the 'offer' routine, which depending on the protocol, will either return an accept/transfer handle, or merely an accept handle, as can be seen in the type of 'offer'. 'handle' is then defined differently depending on the protocol that we are in, and has to inspect what state it is given in protocol P_1 (as it can be accept or transfer), but not in protocol P_2 (as it can only ever be an accept). We can then implement the calling algorithm as follows:

```
main = forever call

call = do
  proto  ← sense
  hdl    ← offer proto
  status ← alert proto hdl
  process proto status

-- Nothing stops me from mixing the protocols here,
-- i.e. returning a status that doesn't correspond
-- to the protocol under use.
alert :: Protocol → AcceptedOrTransfer → IO ProtoStatus
alert protocol hdl = return $
  case protocol of
    P1 → case hdl of
           Accept hdl → S1 1 -- (.. hdl ..)
           Transfer thdl → S1 1 -- (.. thdl ..)
    P2 → case hdl of
           Accept hdl → S2 1 -- (.. hdl ..)
           Transfer thdl → error "Can␣never␣happen"

process :: Protocol → ProtoStatus → IO ()
process protocol s =
  case protocol of
    P1 → case s of
          S1 s1 → return () -- .. s1 ..
          S2 s2 → error "Can␣never␣happen"
    P2 → case s of
          S1 s1 → error "Can␣never␣happen"
          S2 s2 → return () -- .. s2 ..
```

In the definitions of `alert` and `process`, for totality, we had to state the cases which correspond to impossible combinations of protocols and states, according to the specification, even though we know that these will never be called. This is a common problem of using a supertype. There is no way for the compiler to be able to tell us whether the cases that can never happen are indeed unreachable. Moreover, we don't ensure consistency as there's nothing connecting the types of the arguments and the types of the return values, as we see for instance in the type of `alert`. We cannot write a type that enforces that the status returned

corresponds to the protocol under use. The second approach, that of using a tagged union improves the guarantees that the compiler is able to offer. We can thus create a tagged union that will reflect only the possible combinations:

```
data ProtoAccXfer = P1A Handle | P2A Handle | P2T THandle
data Protocol = P1 | P2
data ProtoStatus = S1 Status1 | S2 Status2

main = forever call
sense = undefined
offer = undefined

call :: IO ()
call = do
  proto    ← sense
  protohdl ← offer proto
  status   ← alert protohdl
  process status

-- Nothing stops us from mixing the protocols here,
-- i.e. returning a status that doesn't correspond
-- to the protocol under use.
alert :: ProtoAccXfer → IO ProtoStatus
alert protohdl = return $
  case protohdl of
    P1A hdl → S1 1 -- (.. hdl ..)
    P2A hdl → S2 1 -- (.. hdl ..)
    P2T thdl → S2 1 -- (.. thdl ..)

process :: ProtoStatus → IO ()
process s =
  case s of
    S1 s1 → return () -- .. s1 ..
    S2 s2 → return () -- .. s2 ..
```

The `ProtoAccXfer` type is however, slightly unnatural, as it is effectively reifying all alternatives as data. Also, even though we now don't have any unnecessary branches, we still don't prevent unsound mixing of the protocols. GADTs help in enforcing the invariants that we

want, and indeed we can write more restrictive types that enforce it:

```
{-# LANGUAGE GADTs #-}
import Control.Monad

type Handle = Int
type THandle = Int
type Status1 = Int
type Status2 = Int

data P1
data P2

data PS p where
  PS1 :: PS P1
  PS2 :: PS P2

data AcceptedOrTransfer p where
  Accept   :: Handle → AcceptedOrTransfer p
  Transfer :: THandle → AcceptedOrTransfer P1

data ProtoStatus p where
  S1 :: Status1 → ProtoStatus P1
  S2 :: Status2 → ProtoStatus P2

main :: IO ()
main = forever call

sense = undefined
offer = undefined

call :: IO ()
call = do
  proto  ← sense
  hdl    ← offer proto
  status ← alert proto hdl
  process proto status
  return ()
```

```
-- Mixing protocols is a type error!
-- But the type can't be inferred.
alert :: PS p → AcceptedOrTransfer p
      → IO (ProtoStatus p)
alert protocol hdl = return $
  case protocol of
    PS1 → case hdl of
          Accept hdl → S1 1
          Transfer thdl → S1 1
    PS2 → case hdl of
          Accept hdl → S2 1

process :: PS p → ProtoStatus p → IO ()
process protocol s =
  case protocol of
    PS1 → case s of
            S1 s1 → return ()
    PS2 → case s of
            S2 s2 → return ()
```

However, since complete type inference for unrestricted GADTs is undecidable, the type of alert has to be explicitly written down and can't be inferred. The explicit use of singletons and type-level witnesses also makes the program fairly tricky to understand, and a lot more complex than the previous solutions. However, we have regained safety. This example also shows why we chose to specify the operational semantics for CDT directly, rather than embedding them in Haskell. Deriving GADT constraints using singletons as above would be very complex, as we would have to derive the appropriate singletons, GADTs and constraints in a sound way.

CDT allows us to provide this exact same behaviour, but with a more convenient syntax. We are able to use exactly the simpler encoding that we had in our naive initial implementation. However, we retain all of the guarantees of the GADT version: it is impossible to return a value corresponding to another protocol without this being reflected in the type and the compiler guarantees that we only test for the cases that do happen. Moreover, since our type system is designed specifically around enforcing these types of constraints, we retain complete type inference, which is very helpful when the constraints get more complicated.

```
data AcceptedOrTransfer = Accept Handle | Transfer THandle
```

```
data Protocol = P1 | P2

main = loop protocol[call]

offer : Case protocol of
  { P1 ⇒ IO AcceptedOrTransfer ; P2 ⇒ IO Handle } ↓ {protocol} ! ∅
offer = ..

call = do
  hdl ← offer
  status ← alert hdl
  process status

-- I can't mix the protocols here.
-- That will produce a type error.
alert : Case protocol of
  { P1 ⇒ AcceptedOrTransfer → IO Status1
  ; P2 ⇒ Handle → IO Status2} ↓ {protocol} ! ∅
alert = ccase protocol of {
  P1 ⇒ λhdl. case hdl of
         Accept hdl ⇒ S1 .. hdl ..
         Transfer thdl ⇒ S1 .. thdl .. ;
  P2 ⇒ λhdl. S2 .. hdl ..
}

process : Case protocol of { P1 ⇒ Status1 → IO ()
                           ; P2 ⇒ Status2 → IO () } ↓ {protocol} ! ∅
process s = ccase protocol of {
  P1 ⇒ .. s1 ..
  P2 ⇒ .. s2 ..
}
```

Any other definition of 'alert', that for instance, doesn't check what state it is in when using protocol P_1, will be detected by the compiler and produce a type error.

7.4 Discussion

The telephony example is a practical application of context-dependent types that showcases the relevance and practical applicability of the techniques described in this thesis. It is a context-aware application, where common approaches to developing context-aware applications fail in terms of either safety or flexibility. Dynamically typed frameworks make it hard to reason about the program modularly, as there are several type constraints that have to be kept in the mind of the developer and manually checked, or they will result in runtime errors. As mentioned in Chapter 3, all of the current context-oriented programming approaches do not allow context-dependent types. Haskell, with its approximation of dependent types, was able to provide an approximation of the safety and flexibility we required, however the encoding was fairly complicated and hard to understand.

Dependently typed languages have limited practical applicability, as they restrict computational power, to ensure termination of type checking, and struggle with type inference. As we mentioned in Chapter 1, we could have taken the approach of attempting to mitigate the impact of these fundamental limitations. This would give us the advantage that all the elements of the language would be first-class and able to be manipulated within the language. However, we would also have to limit the operations available in order to ensure that type inference is possible. This would pollute the syntax of the DSL with additional combinators to guide inference, and would also rely on additional type-level machinery that could be unintelligible for the user of the DSL (for instance in terms of error messages).

We thus chose to start with a well-understood base and add the type dependencies needed to express context-dependent types. CDT allows us to encode the domain-specific constraints naturally, by ensuring that values from the two versions of the program, for each protocol, are kept separate when they need to be. In this way we get the safety guarantees that we had in the Haskell GADT encoding but with a simpler syntax that reflects the domain more closely. Moreover, we retain type inference as the only dependency between the contextual protocol and the return types of functions that we can have is modal adaptation. Both in Haskell's GADT example and in dependently typed languages, we would be able to express arbitrary dependencies in the types. For instance, 'alert' in the GADT example would admit a non-dependent type as well:

```
alert :: PS p → AcceptedOrTransfer p
      → IO (ProtoStatus p')
```

As we alluded to, this problem would have to be solved by resorting to additional combinators at the value level, to inform the type level, adding syntactic noise to the DSL. Moreover,

we believe this restriction in flexibility is semantically meaningful in the domain that we are targeting, that of context-aware adaptation. Adopting it allows us to restrict the types of dependency on context that are allowed, and thereby be able to provide stronger guarantees for those.

However, there are several limitations that come through the foundational approach that we took. For instance, as we discussed, it is unclear how to manage the interaction of polymorphism with alternatives, and we leave that for future work. Another restriction that our system poses is in enforcing a strict structure for contextual values through a fixed ontology of labels. We argued for it in the introduction as a meaningful design goal. This was in terms of separation of concerns and reusability of contextual structures across applications. However, it is interesting to examine what one could gain by allowing labels to be first class. Firstly, we could define ad-hoc contexts and context processors within the program without having to defer them all to top-level context label definitions (of the form `context l := e`). One could then imagine the possibility of abstracting away skeletons of modal contextual adaptation on several labels, based on suitable functions that normalise the values of the labels. For instance:

$$\textbf{ccase}\ (\mathit{toHighLow}\ l1)\ \textbf{of}\ \{\mathit{High} \Rightarrow\ \textbf{ccase}\ (\mathit{toTrueFalse}\ l2)\ \textbf{of}\ \{\mathit{True} \Rightarrow \ldots ; \mathit{False} \Rightarrow \ldots\}; \mathit{Low} \Rightarrow \ldots\}$$

This possibility would complicate the type system, as we would need to be able to encode fuzzier constraints (e.g. a particular computation could require any context that can be turned into High/Low). This could be achieved, for instance, through a Haskell-style type class system.

In particular, this extension would also have an effect on the complexity of the type inference algorithm. Our inference algorithm is purely structural, and thus will be called as many times as there are syntactical constructs in the term. However, it will generate fresh variables and perform unification on them. As is well known, this explosion in the number of variables to unify makes second-order polymorphic lambda calculus typability DEXPTIME-hard [HM91, KTU90]. However, Hindley and Milner's algorithm is usable in practice, because the types of programs that cause an explosion in the number of variables to unify are unlikely to appear in practical programming. The addition of modal adaptation can cause types to grow larger in a similar way. However, due to the way we restrict labels to a fixed ontology, this growth is also fixed. Even in the presence of recursion, we can always resolve any inner repeated occurences of a label. For instance:

$$\textbf{Case}\ l\ \textbf{of}\ \{v_1 \Rightarrow\ \textbf{Case}\ l\ \textbf{of}\ \{v_1 \Rightarrow T_1; \ldots\}; \ldots\} \sim\ \textbf{Case}\ l\ \textbf{of}\ \{v_1 \Rightarrow T_1; \ldots\}$$

In this way, our constructs should not add any additional type variable explosion problems that are not already present in the polymorphic lambda calculus. However, the addition of parameterised contexts as discussed before would make these problems more complex, as the same term can be instantiated with different labels in different places. This can be particularly problematic in the presence of recursion, as types can grow arbitrarily large by repeated contextual adaptation on fresh labels. If we continue the analogy we made previously to type and effect systems, in particular region systems, the addition of parameterised contexts would correspond to polymorphic regions. Type inference in the presence of unrestricted polymorphic recursion is known to be undecidable [TB98]. Inference would then require techniques similar to Tofte and Birkedal's region inference algorithm [TB98], which enforces a bound on variables that is guaranteed to be sufficient for expressing the output of inference. However, the restricted algorithm may not yield the principal type, and is technically much more elaborate. As we argued before, we believe that the flexibility that is afforded by this parameterisation is not essential for the use cases we have in mind for the language, especially given the complications it would pose for type inference.

CHAPTER 8

Conclusion

8.1 Summary of Thesis Achievements

This thesis presented a comprehensive analysis of context-awareness through a functional programming perspective, in particular involving types. We outlined a series of design goals that were in our opinion under-represented in existing systems for context-aware program development attempts at solving the context-awareness problem in the research literature.

Composability is essential for the development of reusable software. Indeed, reusability is discouraged if in order to use a context-aware software module, the programmer has to restructure the application to integrate the runtime that is used by a particular framework. We presented a model for context-awareness that is inspired by functional programming and reflected upon what composability means in that setting. Based on this model, we defined a notion of context-dependent composition and showed how it could be encoded as an extension of the lambda calculus. We provided both translation based semantics and direct semantics. The former is useful for integrating these extensions as a library/EDSL in current functional programming languages, while the latter is useful for reasoning about the metatheoretical properties of the extensions.

We then incorporated the model in a practical functional programming language, Haskell, and explored the design space of context dependent values. Namely, we explored how to enrich the knowledge base representation to lift the rigid nesting of contextual types that was enforced by the translation based semantics. In order to do this, we relied on Haskell's type directed translation for type classes and the HList library for type-level lists. This allowed us to provide more expressive context provision operators, that did not require any specific ordering or nesting of the values, but instead would assign them by type. This has

	Light	*Comp*	*Ont*	*CDT*	*Safe*	*Inf*
HCONTEXT (Chapter 5)	✓	✓	✓	✗	✓	✓
λ_{env} (Chapter 6)	✓	✓	✗	✓	✓	✓
CDT (Chapter 7)	✓	✓	✓	✓	✓	✓

Table 8.1: Analysis of the proposed solutions with regards to the design goals.

the obvious problem that we cannot have several different types of context with the same type, as would be the case with, for instance, Location. In order to do this, we developed type-level synonyms for basic types, that were tagged with domain-specific identifiers, such as individuals and roles.

We then implemented a model for abstracting away context provision into a runtime layer. This layer again exploits Haskell's type directed translation to produce the necessary sensor querying code for the context dependencies of the computation at hand. This allowed us to provide an extremely concise implementation of a classical context-aware application, the location-based shop listing application. We believe that by allowing the developer to perform typical context-aware operations very easily, we can enable the emergence of much more complex applications than have been seen to date.

The other facet of context-awareness which we explored is related to the often mentioned "device as context" idea. Even though this is conceptually elegant, in practice it has severe problems, in that different devices will have different data representations and thus use different types. In order to mitigate this problem, we argue that context-dependent types are an important feature. We thus design an extension of the lambda calculus featuring context-dependent types, as modal global alternatives. We then prove type soundness of the system, and design a sound and complete type inference algorithm for it. This enables easier practical usage of the language. Most of the metatheory is implemented in the Coq proof assistant, thus providing higher reliability guarantees on the properties proven and allowing us to derive portions of the system's implementation.

We then tie everything together in a high-level programming language incorporating the contextual representation techniques from the EDSL and context dependent types and show how we can provide a better solution for the implementation of a real world application from the telephony industry than current state of the art programming languages. For the sake of comparison, in Table 8.1 we present an analysis of all of our proposed approaches with regards to the design goals outlined in the introduction. As we mentioned in section 5.5, HCONTEXT fulfils all of the desired design goals except for context-dependent types. However, it presents an interesting practical EDSL for the development of context-aware applications. λ_{env} was then a simplified calculus, to analyse the addition of context-dependent types to

a functional programming language. We did not consider the definition of ontologies or context runtimes, deferring that to an oracle. We had also omitted practical programming features such as data types, which are essential for practical usage, but which are not part of our design goals as they do not pertain specifically to context-awareness and are somewhat orthogonal. Finally, we used the calculus as the basis for a practical programming language, by adding those practical programming language features and a way to define ontologies and context sources, thus fulfilling all the original design goals.

8.2 Applications

The widespread availability of computing devices featuring various sensors provides very interesting opportunities to provide richer experiences to the users; that make more sense given their current circumstances. These opportunities bring with them more complexity and pitfalls. By providing a sound model of how context influences applications we believe that we can help development platforms of the future in providing easier and safer abstractions for context-awareness.

Indeed, we believe that the main application of this work is to be incorporated in development platforms, allowing developers to create libraries that properly abstract typical context-awareness features. Moreover, by incorporating context-dependencies at the type level we can allow the device, its subcomponents and environment to truly be treated as a context, thus paving the way for portable executables and reducing the barrier to application adoption. By incorporating these variations at the library level, developers can effortlessly implement portable applications. For example, we can imagine a context-aware operating system, whereby the necessary platform/peripheral variability is encoded in a safe way that is validated by the compiler.

In a future that is increasingly multi-platform, we believe that designing technologies to enable safe portability is essential.

8.3 Future Work

8.3.1 Full formal treatment of CDT and type inference

As mentioned in Section 7.1.1, we have implemented in CDT several extensions to λ_{env} that are necessary for practical use as a programming language. However, even though they are

fairly conservative, we have not proven that type soundness, and soundness of inference hold after adding these properties. This should be done to ensure correctness, and to have an implementation of the language that is mechanically proven to be correct. Moreover, we have only conjectured that completeness for type inference holds, justifying how we think we have handled the only source of non-determinism that the (syntax-directed) type system introduces. However, to deliver stronger correctness guarantees, a full formal proof would be important.

8.3.2 Polymorphism

The decision to add polymorphism at the top level is not trivial and has some subtle consequences. In particular, the interaction of polymorphism with the $\sim$ relation is worth mentioning. For example, if we consider $\Gamma_0 = \{x_a : T_a, x_b : T_b, id : \forall X.X \to X\}$, it is interesting to examine the following type derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{id : \forall X.X \to X \in \Gamma_0}{\Gamma_0 \vdash id : \forall X.X \to X ! \varnothing}}{\Gamma_0 \vdash id : T_a \to T_a ! \varnothing} \quad T_a \to T_a \sim \textbf{Case } l \textbf{ of } \{...\}}{\Gamma_0 \vdash id : \textbf{Case } l \textbf{ of } \{v_1 \Rightarrow T_a \to T_a; v_2 \Rightarrow T_a \to T_a\} ! \varnothing} \quad \dfrac{...}{\Gamma_0 \vdash (\textbf{ccase } l \textbf{ of } \{v_1 \Rightarrow x_a; v_2 \Rightarrow x_a\}) : ...}}{\Gamma_0 \vdash id(\textbf{ccase } l \textbf{ of } \{v_1 \Rightarrow x_a; v_2 \Rightarrow x_a\}) : T_a ! \varnothing}$$

This example shows how one could generalise the type of a polymorphic identity function, as will be done for instance by type inference, and then instantiate that type to match that of the argument. We could not modify this example in order to type the following (arguably meaningful) term:

$$(\lambda x.x)(\textbf{ccase } l \textbf{ of } \{v_1 \Rightarrow x_a; v_2 \Rightarrow x_b\})$$

It is interesting to explore the ways in which we could modify the system to include this term. One potential idea would be to think of alternatives as having their own separate variable environments. We could then have alternatives at the top level of types, and universal quantification within alternatives. This would however require us to distinguish between alternatives that only contain simple types, and those that contain polymorphic types, which cannot be done syntactically, as in Hindley-Milner. This might require the addition of a kind system, to distinguish between these alternatives, along with kinding constraints on all rules.

The other possibility is to modify the equivalence relation to take into account polymorphic types. For instance, we could define $\sim_\sigma$ as an alternative to $\sim$ requiring that all universally quantified variables are made fresh in each alternative. For instance, consider:

$$\forall X.X \to X \sim_\sigma \forall X.\forall X'.\,\textbf{Case}\ l\ \textbf{of}\ \{v_1 \Rightarrow X \to X; v_2 \Rightarrow X' \to X'\}$$

Using this relation in the Subst rule, should allow us to preserve polymorphism at the top-level and still type the problematic term. However, we leave developing such a type system for future work. Polymorphism in the present calculus is thus available only as long as the application sites in all alternatives have the same type. This is a limited form of polymorphism, but is already useful, as can be seen in the example derivation above.

8.3.3 Quality of Context

The context sources as we have considered them so far are fairly primitive. We can only fetch the current context and manipulate it. Modelling context sources in a richer way has been one of the main focus of context-awareness research in the ubiquitous computing community [HM06]. It would be interesting to see if we can provide a more formal grounding for some of those techniques using techniques similar to the ones presented in this thesis.

In particular, when combining multiple sources with different qualities of context (e.g. accuracy and sampling rate), we could algebraically derive the quality of context of the newly defined source.

8.3.4 Historical data and timing

In the current system, we assume that we are interested only in the current context and will make decisions based on it. While we can use side-effects to store historical values and process them, another approach that would be more aligned to the design goals of this thesis would be to attempt to provide natural language primitives for fetching and manipulating historical data.

One possible application of this would be to automatically derive the code that does the maintenance of historical contextual data stores. This is particularly interesting in constrained devices, where we could, for instance, merely store the past values that we are sure are going to be used in the future, thus saving memory and reducing lookup times. By incorporating the time dependencies into the types as well, and reasoning about how computations with time dependencies compose, we can do this in a way that is provably sound.

8.4 Final Remarks

Current practical context-aware applications have so far been developed in languages that do not appropriately model context-awareness, thus hurting their modularity and composability and making it harder to understand and use context-aware behaviour. In this thesis we have shown that it is possible and advantageous to provide safer development environments, where context-aware behaviour is proven correct by the compiler and thus easier to implement and manage.

We believe that context-awareness will become increasingly important in development environments of the future. We hope that the work presented in this thesis helps pave the way for the development of environments featuring even more complex abstractions and correctness guarantees, thus enabling the manageable development of intricate context-aware behaviour.

Bibliography

[AMM05] T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. *Manuscript, available online*, 2005.

[Atk09] R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335, June 2009.

[Awo06] S. Awodey. *Category theory*. Oxford University Press, USA, 2006.

[Bar92] H.P. Barendregt. Lambda calculi with types. *Handbook of logic in computer science*, 2:117–309, 1992.

[Bar98] H.P. Barendregt. *The lambda calculus: its syntax and semantics*. Elsevier, 1998.

[Bar05] J.E. Bardram. The Java Context Awareness Framework (JCAF)–a service infrastructure and programming framework for context-aware applications. *Pervasive Computing*, pages 98–115, 2005.

[BBH+09] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 2009.

[BCM+07] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider. *The description logic handbook*. Cambridge University Press New York, NY, USA, 2007.

[BDM96] R. Bird and O. De Moor. The algebra of programming. *Deductive Program Design*, 1996.

[BDR07] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

[BH91] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In *International Joint Conference on Artificial Intelligence*, pages 452–457, 1991.

[BKM+12] N. Baccour, A. Koubâa, L. Mottola, M.A. Zúñiga, H. Youssef, C.A. Boano, and M. Alves. Radio link quality estimation in wireless sensor networks: a survey. *ACM Transactions on Sensor Networks (TOSN)*, 8(4):34, 2012.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[Bru72] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[BS09] J.C. Baez and M. Stay. Physics, topology, logic and computation: a Rosetta Stone. *New Structures for Physics, B. Coecke, Ed. Springer Lecture Notes in Physics.*, pages 1–73, 2009.

[CCC+06] D. Chalmers, M. Chalmers, J. Crowcroft, M. Kwiatkowska, R. Milner, E. O'Neill, T. Rodden, V. Sassone, and M. Sloman. Ubiquitous computing: Experience, design and science, 2006.

[Cha10] Matthew Chalmers. A population approach to ubicomp system design. *Proceedings of the 2010 ACM-BCS Visions of Computer Science Conference*, pages 1:1–1:12, 2010.

[Coq02] The Coq. The Coq Proof Assistant : Reference Manual : Version 7.2. Technical Report RT-0255, INRIA, February 2002.

[CS09] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *International Workshop on Context-Oriented Programming*, COP '09, pages 10:1–10:6, New York, NY, USA, 2009. ACM.

[DA] A.K. Dey and G.D. Abowd. Context Toolkit User's Guide, `http://www.cc.gatech.edu/fce/contexttoolkit/documentation/UserGuide.html`.

[DA00a] A.K. Dey and G.D. Abowd. The context toolkit: Aiding the development of context-aware applications. In *Workshop on Software Engineering for wearable and pervasive computing*, pages 431–441. Citeseer, 2000.

[DA00b] A.K. Dey and G.D. Abowd. Towards a better understanding of context and context-awareness. In *CHI 2000 workshop on the what, who, where, when, and how of context-awareness*, pages 304–307, 2000.

[Dam84] Luis Manuel Martins Damas. *Type assignment in programming languages.* PhD thesis, University of Edinburgh, 1984.

[DAS01] A.K. Dey, G.D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2):97–166, 2001.

[DDDCG02] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: Fickle &par. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(2):153–191, 2002.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM New York, NY, USA, 1982.

[Ell09] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

[FGA08] F. Foukalas, V. Gazis, and N. Alonistioti. Cross-layer design proposals for wireless mobile networks: a survey and taxonomy. *Communications Surveys & Tutorials, IEEE*, 10(1):70–85, 2008.

[GHC11] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.0.3*, 2011.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[HCN08] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of object technology*, 7(3), 2008.

[HHJW07] P. Hudak, J. Hughes, S.P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–55. ACM, 2007.

[HHPJW96] C.V. Hall, K. Hammond, S.L. Peyton Jones, and P.L. Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):138, 1996.

[HIM11] R. Hirschfeld, A. Igarashi, and H. Masuhara. Contextfj: a minimal core calculus for context-oriented programming. In *Proceedings of the 10th international workshop on Foundations of aspect-oriented languages*, pages 19–23. ACM, 2011.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[HM91] Fritz Henglein and Harry G. Mairson. The complexity of type inference for higher-order lambda calculi. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 119–130, New York, NY, USA, 1991. ACM.

[HM05] Markus C. Huebscher and Julie A. McCann. Using real-time dependability in adaptive service selection. In *Joint International Conference on Autonomic and Autonomous Systems 2005 / International Conference on Networking and Services 2005 (ICAS/ICNS 2005), 23-28 October 2005, Papeete, Tahiti*, page 76. IEEE Computer Society, 2005.

[HM06] M.C. Huebscher and J.A. McCann. An adaptive middleware framework for context-aware applications. *Personal and Ubiquitous Computing*, 10(1):12–20, 2006.

[Hor98] I. Horrocks. The fact system. *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 307–312, 1998.

[HPSVH03] I. Horrocks, P.F. Patel-Schneider, and F. Van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Web semantics: science, services and agents on the World Wide Web*, 1(1):7–26, 2003.

[HS01] I. Horrocks and U. Sattler. Ontology reasoning in the shoq (d) description logic. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 199–204. Citeseer, 2001.

[HS04] I. Horrocks and U. Sattler. Decidability of shiq with complex role inclusion axioms. *Artificial Intelligence*, 160(1-2):79–104, 2004.

[Hue97] G. Huet. The zipper. *Journal of Functional Programming*, 7(05):549–554, 1997.

[Hug89] J. Hughes. Why functional programming matters. *The computer journal*, 32(2):98, 1989.

[Hug00] J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111, 2000.

[HYC08] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, 2008.

[IBM06] IBM. An architectural blueprint for autonomic computing. *IBM White Paper*, 2006.

[IHM12] A. Igarashi, R. Hirschfeld, and H. Masuhara. A type system for dynamic layer composition. *FOOL'12*, 2012.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[JJM97] S.P. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *In Haskell Workshop*, 1997.

[Jon99] Mark P Jones. Typing haskell in haskell. In *Haskell workshop*, volume 7, 1999.

[Jon00] M. Jones. Type classes with functional dependencies. *Programming Languages and Systems*, pages 230–244, 2000.

[Jon03] S.P. (ed.) Jones. *Haskell 98 language and libraries: the revised report*. Cambridge Univ Pr, 2003.

[KHS06] O. Kutz, I. Horrocks, and U. Sattler. The even more irresistible sroiq. *Proc. KR*, pages 68–78, 2006.

[KJS10] O. Kiselyov, S.P. Jones, and C. Shan. Fun with type functions. *Reflections on the Work of CAR Hoare*, pages 301–331, 2010.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell 2004: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[Kor03] Korpipää, P. and Mäntyjärvi, J. and Kela, J. and Keränen, H. and Malm, E.J. Managing context information in mobile devices. *IEEE pervasive computing*, 2(3):42–51, 2003.

[KTU90] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Ml typability is dexptime-complete. In *Proceedings of the Fifteenth Colloquium on CAAP'90*, CAAP '90, pages 206–220, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[KV06] J. Kabanov and V. Vene. Recursion schemes for dynamic programming. In *Mathematics of Program Construction*, pages 235–252. Springer, 2006.

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[LG88] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57. ACM, 1988.

[Lia99] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[LLMS00] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 108–118, New York, NY, USA, 2000. ACM.

[LS86] J. Lambek and P.J. Scott. *Introduction to higher order categorical logic*, volume 164. Cambridge University Press, 1986.

[LS97] FW Lawvere and S.H. Schanuel. *Conceptual mathematics: a first introduction to categories*. Cambridge Univ Pr, 1997.

[LS00] H. Lieberman and T. Selker. Out of context: Computer systems that adapt to, and learn from, context. *IBM Systems Journal*, 39(3.4):617–632, 2000.

[Luk08] T. Lukasiewicz. Expressive probabilistic description logics. *Artificial Intelligence*, 172(6-7):852–883, 2008.

[M+10] S. Marlow et al. Haskell 2010 language report, 2010.

[Mac71] S. Mac Lane. Categories for the Working Mathematician. 1971. *Graduate Texts in Mathematics*, 1971.

[Mai07] G. Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

[McB01] C. McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, 2001.

[McB02] Conor McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, July 2002.

[McB05] C. McBride. Epigram: Practical programming with dependent types. *Lecture Notes in Computer Science*, 3622:130–170, 2005.

[McK06] J. McKinna. Why dependent types matter. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, volume 33, page 1. ACM, 2006.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[MGH+08] B. Motik, B.C. Grau, I. Horrocks, Z. Wu, O.A. Fokoue, I.B.M.C. Lutz, et al. Owl 2 web ontology language: Profiles. *W3C Working Draft*, 2008.

[MGH+12] B. Motik, B.C. Grau, I. Horrocks, B. Parsia, and U. Sattler. Owl 2 web ontology language direct semantics. *Latest version available at http://www. w3. org/TR/owl2-semantics*, 2012.

[Mil06a] R. Milner. Pure bigraphs: Structure and dynamics. *Information and computation*, 204(1):60–122, 2006.

[Mil06b] R. Milner. Ubiquitous computing: shall we understand it? *The Computer Journal*, 49(4):383, 2006.

[MM10] P. Martins and J.A. McCann. ajME: making game engines autonomic. In *Proceedings of the 3rd International Conference on Fun and Games*, pages 48–57. ACM, 2010.

[MME12] Pedro Martins, Julie McCann, and Susan Eisenbach. The Environment as an Argument. In *Fourteenth International Symposium on Practical Aspects of Declarative Languages*, volume 7149 of *Lecture Notes in Computer Science*, January 2012.

[Mog91] E Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, July 1991.

[MP07] C. McBride and R. Paterson. Applicative programming with effects. *Journal of functional programming*, 18(01):1–13, 2007.

[MPSH08] B. Motik, P.F. Patel-Schneider, and I. Horrocks. Owl 2 web ontology language: Structural specification and functional-style syntax. *W3C Working Draft, W3C*, 2008.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 35. Citeseer, 2007.

[OSG08] B. O'Sullivan, D.B. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, Inc., 2008.

[PB08] Davy Preuveneers and Yolande Berbers. Internet of things: a context-awareness perspective. *The Internet of Things. From RFID to the Next-generation Pervasive Network Systems, Auerbach Publications, Taylor & Francis Group*, pages 287–307, 2008.

[PCG+10] Benjamin C Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjoberg, and Brent Yorgey. Software foundations. *Course notes, online at http://www. cis. upenn. edu/˜ bcpierce/sf*, 2010.

[PGPM99] H.S. Pinto, A. Gómez-Pérez, and J.P. Martins. Some issues on ontology integration. In *IJCAI-99 workshop on ontologies and problem-solving methods (KRR5)*. Citeseer, 1999.

[Pie02] Benjamin C Pierce. *Types and programming languages*. The MIT Press, 2002.

[RC00] Manuel Roman and Roy H. Campbell. Gaia: enabling active spaces. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 229–234, New York, NY, USA, 2000. ACM.

[Sco93] D.S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1-2):411–440, 1993.

[SDA99] D. Salber, A.K. Dey, and G.D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, pages 434–441. ACM, 1999.

[SE09] Matthew Sackman and Susan Eisenbach. Safely Speaking in Tongues: Statically Checking Domain Specific Languages in Haskell. In *LDTA'09*, March 2009.

[SGP11] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A programming paradigm for autonomic systems. *arXiv preprint arXiv:1105.0069*, 2011.

[SJ02] T. Sheard and S.P. Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.

[SK10] Curt Sampson and Lars Kotthoff. Mhailist: Haskell mailing list manager. `http://hackage.haskell.org/package/Mhailist-0.0`, April 2010.

[SLP04] T. Strang and C. Linnhoff-Popien. A context modeling survey. In *Workshop on Advanced Context Modelling, Reasoning and Management as part of UbiComp*. Citeseer, 2004.

[SNO+07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: effective tool support for the working semanticist. *SIGPLAN Not.*, 42(9):1–12, October 2007.

[Sø90] Søndergaard, H. and Sestoft, P. Referential transparency, definiteness and unfoldability,. *Acta Informatica*, 27(6):505–517, 1990.

[SPJCS08a] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. *SIGPLAN Not.*, 43:51–62, September 2008.

[SPJCS08b] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 51–62, New York, NY, USA, 2008. ACM.

[Ste02] Don Stewart. Multi-paradigm Just-in-time Compilation. Master's thesis, The University of New South Wales, School of Computer Science and Engineering, November 2002.

[Str02] U. Straccia. Reasoning within fuzzy description logics. *Journal of Artificial Intelligence Research*, 14:147–176, 2002.

[SW09] Hedda R. Schmidtke and Woontack Woo. Towards ontology-based formal verification methods for context aware systems. In *Pervasive '09: Proceedings of the 7th International Conference on Pervasive Computing*, pages 309–326, Berlin, Heidelberg, 2009. Springer-Verlag.

[Swi08] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(04):423–436, 2008.

[TB98] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):724–767, 1998.

[Tul96] M. Tullsen. Compiling haskell to java. *Technical Report, Department of Computer Science–Yale University, New Heaven*, 1996.

[Tur04] D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, Jul 2004. `http://www.jucs.org/jucs_10_7/total_functional_programming`.

[UV05] T. Uustalu and V. Vene. Signals and comonads. *J. of Univ. Comput. Sci*, 11(7):1310–1326, 2005.

[Vau08] Jeff Vaughan. A proof of correctness for the hindley-milner type inference algorithm, 2008.

[vB01] S. van Bakel. Type Systems for Programming Languages (Course Notes). 2001.

[Wad89] P. Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM New York, NY, USA, 1989.

[Wad93] P. Wadler. Monads for functional programming. *NATO ASI Series F Computer and Systems Sciences*, 118:233–233, 1993.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[Wei95] M. Weiser. The Computer for the 21st Century. *Scientific American*, 272(3):78–89, 1995.

[WF92] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.

[YCDN07] Juan Ye, Lorcan Coyle, Simon Dobson, and Paddy Nixon. Ontology-based models in pervasive computing systems. *Knowl. Eng. Rev.*, 22(4):315–347, 2007.

[YSD+09] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (Oakland'09)*, IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009.

[YWC+12] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In

Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.

[ZKEH06] Antoine Zimmermann, Markus Krötzsch, Jérôme Euzenat, and Pascal Hitzler. Formalizing ontology alignment and its operations with category theory. In *Proceeding of the 2006 conference on Formal Ontology in Information Systems*, pages 277–288, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.

APPENDIX A

Embedding $\lambda_{\downarrow}$ in the STLC

In this section we will present how to embed the type system of $\lambda_{\downarrow}$ in the STLC by using products instead of union. This embedding is useful as it shows us how we can use these techniques in already existing functional programming languages, as was shown in Section 5.

For the purposes of the embedding, it is useful to identify contextual types as language types. Moreover, we can map union to pairing the sets. This is not ideal as we lose associativity, but we will later see ways in which we can alleviate this restriction. In the meantime, it keeps the embedding simple. Thus, we modify the above calculus slightly, to form the $\lambda_{\downarrow\star}$ system, which we present in Figure A.1. We override the metavariable e to now range over the extended expressions, for simplicity.

Terms of the form $?t$ are used to retrieve contextual information by type, and introduce a contextual dependency on t. Contextual dependencies can be satisfied by providing a term of the appropriate type, with the syntax $e \star e$. For example, assuming types *Location* and *Double*, a term *loc_home* of type *Location* and a term *distance* of type $Distance \to Distance \to Double$ we could have the following typing, for a (contrived) term that calculates the distance between a *User* and their home location (*loc_home*):

$$\vdash ((\lambda x.\ distance\ ?Location\ x)\ loc\_home) : Double \downarrow Location$$

This illustrates how the contextual dependency is propagated through the lambda, to the top-level. If, for instance, the user were at home, we could then discharge the contextual dependency by providing the location value (in this case, *loc_home*):

$e \in CExpr ::= x \mid \lambda x.e \mid e\,e \mid ?t \mid e \star e$

$\boxed{\Gamma \vdash e : t \downarrow t'}$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t \downarrow 1}\ \textsc{CAx}$$

$$\frac{\Gamma, x : t \vdash e : t' \downarrow t_c}{\Gamma \vdash \lambda x.e : t \to t' \downarrow t_c}\ \textsc{CLam}$$

$$\frac{\Gamma \vdash e_1 : t \to t' \downarrow t_c \quad \Gamma \vdash e_2 : t \downarrow t'_c}{\Gamma \vdash e_1\,e_2 : t' \downarrow t_c \times t'_c}\ \textsc{CApp}$$

$$\frac{}{\Gamma \vdash ?t : t \downarrow t}\ \textsc{CLabel}$$

$$\frac{\Gamma \vdash e_1 : t \downarrow t_c \quad \Gamma \vdash e_2 : t_c \downarrow 1}{\Gamma \vdash e_1 \star e_2 : t \downarrow 1}\ \textsc{CCApp}$$

Figure A.1: Grammar and type system of $\lambda_{\downarrow\star}$.

$$\vdash ((\lambda x.\ distance\ ?Location\ x)\ loc\_home) \star loc\_home : Double \downarrow 1$$

This term should then evaluate to $0 : Double$ assuming such a term. We can define a mapping from $\lambda_{\downarrow\star}$ to the STLC reflecting these semantics, by propagating the contexts outwards as we described, and splitting the context on application. We use the metavariable c here to range over variables in addition to x, to distinguish between contexts and (explicit) variables.

$$\begin{aligned}
[\![x]\!] &= \lambda c.x \\
[\![e_1 e_2]\!] &= \lambda c.([\![e_1]\!](\pi_1 c)([\![e_2]\!](\pi_2 c))), c \text{ fresh in } e_1 \text{ and } e_2 \\
[\![\lambda x.e]\!] &= \lambda c.\lambda x.([\![e]\!]c), c \neq x \text{ and } c \text{ fresh in } e \\
[\![?t]\!] &= \lambda c.c \\
[\![e_1 \star e_2]\!] &= \lambda c.(e_1 e_2), c \text{ fresh in } e_1 \text{ and } e_2
\end{aligned}$$

This will result in the following mapping of typing judgements, which can be easily proven by induction over the typing derivation.

Theorem A.1. *Preservation of typing over translation: If a term e has type t under environment* Γ *for the contextual tree* t_c*, then* $[\![e]\!]$ *has type* $t_c \to t$ *under the same environment, in STLC:*

$$\Gamma \vdash e : t \downarrow t_c \Rightarrow \Gamma \vdash [\![e]\!] : t_c \to t$$

This system allows us to build up context-aware computations and to collect the contextual dependencies in a tree formed of iterated products. In order to use this information we could then provide a way to discharge the contextual obligations. For instance, we introduce a construct that directly translates to regular application in the STLC for that: $e \star e'$, where e' will have to be the contextual values tree. This is a very primitive solution, used here only for the purpose of demonstration. In section 5 we will replace this term by a smart projection term πt which then undergoes a type-directed translation to pick the right path for that label. This is paired with a smart scheme to remove duplicates from the tree. The type-directed translation scheme is interesting as it can be embedded directly into any functional programming language that features type classes, such as Haskell [M$^+$10].

APPENDIX B

Mechanized metatheory of λ_{env}

In order to ensure a higher level of reliability in the formal results, we have encoded the main formal systems presented in the previous chapters in the Coq theorem prover. We have then encoded the necessary properties as theorems in the language and developed proof terms witnessing their validity. In this chapter we present this encoding as well as some of the more interesting techniques employed in the proof scripts.

B.1 Workflow

In order to ensure a consistent presentation, we used ott [SNO+07] to write up the formal definition of the systems. We then used this to generate the LaTeX output seen in the body of the thesis, as well as Coq definitions corresponding to the systems defined. After proving the results presented in the thesis, we then annotated the Coq scripts with coqdoc comments, and used coqdoc to generate the inline proofs seen in the body of the thesis. The proofs presented in this appendix will thus be comprised of the same proofs, but interspersed with the coq proof script that is witness to the results presented.

B.2 Logic

We are going to assume classical logic:

```
Require Coq.Logic.Classical_Prop.
Lemma excluded_middle : ∀ (P : Prop), P ∨ ~P.
```

We also assume functional extensionality.

```
Require Export Coq.Logic.FunctionalExtensionality.
```

Some useful obvious properties:

```
Lemma demorgan : ∀ p q, ~ (p ∨ q) → ~ p ∧ ~ q.
Lemma contrapositive : ∀ (p q : Prop),
  (p → q) → ( ~q → ~p).
```

B.3 Encodings

B.3.1 Sets

In order to encode (finite) sets within the calculus of constructions we use polymorphic cons lists, defined (inductively) in the Coq standard library as follows:

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A → list A → list A.
```

This encoding is supported by the Coq standard library, in the `Coq.Lists.ListSet` module.

```
Require Export Coq.Lists.ListSet.
```

The obvious problem with this encoding is that we do not have an isomorphism between sets and lists of the elements, because when using a list duplicate elements are allowed and order matters. We sidestep both these issues by careful usage of the utility functions provided in the library. Firstly, we use the order irrelevant predicate *Inxs*, stating that the element x is in the set-as-list s, $x \in s$, as usual. Secondly, instead of set equality, we use an equivalence relation on lists, *equivlistA*, which is defined in `Coq.Lists.SetoidList` as:

```
Variable A : Type.
Variable eqA : A → A → Prop.
Definition equivlistA l l' := ∀ x, InA x l ↔ InA x l'.
```

```
Require Export Coq.Lists.SetoidList.
```

Moreover we define a shorthand notation for equivlistA when we want to use Coq's (Leibniz) equality:

```
Notation "a == b" := (Coq.Lists.SetoidList.equivlistA eq a b) (at level 5).
```

Thus, if we define a representation function $set(s) : List \to Set$, mapping lists to sets of their elements we get the meta-property:

$$set(l) = set(l') \Rightarrow l == l'$$

We define subset similarly:

```
Definition subset {A} a b := ∀ (x : A), In x a → In x b.
Notation "a  b" := ( subset a b) (at level 5).
```

We also define some additional lemmas for sets:

```
Require Export Coq.Lists.List.

Lemma in_set_remove : ∀ {A} (x0 : A) (x : A) s eq,
  x0 ≠ x → ( ~ In x (set_remove eq x0 s)) → ~ In x s.

Definition is_nil {A} (xs : list A) :=
  match xs with
  | nil ⇒ True
  | cons x' xs' ⇒ False
  end.

Lemma is_nil_app : ∀ {A} (xs1 : list A) (xs2 : list A),
  is_nil (app xs1 xs2) → is_nil xs1 ∧ is_nil xs2.

Lemma set_union_add_not : ∀ {A} (a : A) a' l eq,
  ~ In a l → a ≠ a' → ~ In a (set_add eq a' l).

Definition subset_eq {A} (s1 : set A) (s2 : set A) := ∀ (a:A), In a s1 → In a s2.

Lemma subset_eq_nil : ∀ {A} (s1 : set A), subset_eq s1 nil → s1 = nil.

Lemma equivlist_nil_eq : ∀ {A} (l : list A),
  equivlistA eq l nil → l = nil.

Lemma InA_In : ∀ {A} (x:A) s, InA eq x s → In x s.

Fixpoint take {A} (n : nat) (xs : list A) : list A :=
  match (n, xs) with
  | (O, xs') ⇒ nil
  | (S n, cons x xs') ⇒ cons x (take n xs')
  | (S n, nil) ⇒ nil
  end.

Fixpoint drop {A} (n : nat) (xs : list A) : list A :=
  match (n, xs) with
  | (O, xs') ⇒ xs'
  | (S n, cons x xs') ⇒ drop n xs'
  | (S n, nil) ⇒ nil
```

```
  end.
Fixpoint reverse_aux {A} (aux : list A) (xs : list A) :=
  match xs with
  | nil ⇒ aux
  | cons x xs' ⇒ reverse_aux (cons x aux) xs'
  end.
Definition reverse {A} (xs : list A) : list A := reverse_aux nil xs.
Fixpoint rm_all_f {A} (f : A → bool) (xs : list A) : list A :=
  match xs with
  | nil ⇒ nil
  | cons x xs' ⇒ if (f x) then rm_all_f f xs' else cons x (rm_all_f f xs')
  end.
Fixpoint zipWith {A B C} (f : A → B → C) (xs : list A) (xs' : list B) :=
  match (xs, xs') with
  | (cons x1 xs1, cons x2 xs2) ⇒ cons (f x1 x2) (zipWith f xs1 xs2)
  | _ ⇒ nil
  end.
Lemma find_const_false_nil : ∀ {A} {l : list A}, find (fun _ ⇒ false) l = None.
```

We define some more trivial set identities, some of them specialised for sets of labels, for simplicity.

```
Lemma subset_union : ∀ {s1 s2 s3},
  (s1 ∪ s2) ⊆ (s1 ∪ s3) ↔ s2 ⊆ s3.
Lemma subset_union_n : ∀ {s ss},
  In s ss ↔ s ⊆ (⋃i ss).
Lemma union_dist : ∀ {ss s},
  (⋃i ss; s) == (( i ss) ∪ s).
Lemma union_comm : ∀ {s1 s2}, (s1 ∪ s2) == (s2 ∪ s1).
Lemma set_diff_nil_id : ∀ {A eqA s},
  set_diff eqA s (nil : set A) = s.
Lemma set_diff_subset_union : ∀ {l s1 s2},
  (set_diff eqA (s1 ∪ s2) {l}) ⊆ (set_diff eqA s1 {l} ∪ s2).
Lemma set_diff_idem : ∀ {s l},
  (set_diff eqA (set_diff eqA s {l}) {l}) == (set_diff eqA s {l}).
Lemma set_diff_subset : ∀ {l s1 s2},
```

$s1 \subseteq s2 \rightarrow (set\_diff\ eqA\ s1\ l) \subseteq (set\_diff\ eqA\ s2\ l)$.

Lemma *set_diff_dist_union* : ∀ {*s1 s2 l*},
$((set\_diff\ eqA\ s1\ \{l\}) \cup (set\_diff\ eqA\ s2\ \{l\})) == (set\_diff\ eqA\ (s1 \cup s2)\ \{l\})$.

Lemma *set_diff_comm* : ∀ {*s l1 l2*},
$(set\_diff\ eqA\ (set\_diff\ eqA\ s\ l1)\ l2) == (set\_diff\ eqA\ (set\_diff\ eqA\ s\ l2)\ l1)$.

Lemma *set_diff_cons* : ∀ {*s l*},
$(set\_diff\ eqA\ (\{l\} \cup s)\ \{l\}) == (set\_diff\ eqA\ s\ \{l\})$.

Lemma *set_diff_cons2* : ∀ {*s l1 l2*},
$l1 \neq l2 \rightarrow (set\_diff\ eqA\ (\{l1\} \cup s)\ \{l2\}) == (\{l1\} \cup (set\_diff\ eqA\ s\ \{l2\}))$.

Lemma *set_diff_cons3* : ∀ {*s s' l*},
$(set\_diff\ eqA\ s\ (l :: s')) == (set\_diff\ eqA\ (set\_diff\ eqA\ s\ s')\ \{l\})$.

Lemma *set_diff_not_in* : ∀ {*A eqA s s'*} {*a* : *A*},
$In\ a\ s \rightarrow \sim In\ a\ (set\_diff\ eqA\ s'\ s)$.

Lemma *dist_union* : ∀ {*s1 s2 s3*},
$((s1 \cup s2) \cup s3) == ((s1 \cup s3) \cup (s2 \cup s3))$.

Lemma *assoc_union* : ∀ {*s1 s2 s3*},
$((s1 \cup s2) \cup s3) == (s1 \cup (s2 \cup s3))$.

Lemma *idem_union* : ∀ {*s*}, $(s \cup s) == s$.

Lemma *subset_nil* : ∀ {*A*} {*s* : `set` *A*}, $nil \subseteq s$.

Lemma *subset_nil_eq* : ∀ {*A*} {*s* : `set` *A*}, $s \subseteq nil \rightarrow s = nil$.

Lemma *subset_union_l* : ∀ {*s1 s2*}, $s1 \subseteq (s1 \cup s2)$.

Lemma *subset_union_r* : ∀ {*s1 s2*}, $s2 \subseteq (s1 \cup s2)$.

Lemma *subset_union_lr* : ∀ {*s s1 s2*}, $s \subseteq s1 \vee s \subseteq s2 \rightarrow s \subseteq (s1 \cup s2)$.

Lemma *union_eq_nil* : ∀ {*s1 s2*}, $(s1 \cup s2) = nil \rightarrow s1 = nil \wedge s2 = nil$.

Lemma *union_cons_eq_nil* : ∀ {*l s*}, $(\{l\} \cup s) = nil \rightarrow False$.

Lemma *subset_set_add* : ∀ {*c l*}, $c \subseteq (set\_add\ eqA\ l\ c)$.

Lemma *set_add_idem* : ∀ {*l c*},
$(set\_add\ eqA\ l\ (set\_add\ eqA\ l\ c)) == (set\_add\ eqA\ l\ c)$.

Lemma *set_add_comm* : ∀ {*l1 l2 c*},
$(set\_add\ eqA\ l1\ (set\_add\ eqA\ l2\ c)) ==$
$(set\_add\ eqA\ l2\ (set\_add\ eqA\ l1\ c))$.

Lemma *set_add_union* : ∀ {*l c1 c2*},

((*set_add eqA l c1*) ∪ (*set_add eqA l c2*)) ==
(*set_add eqA l* (*c1* ∪ *c2*)).

`Lemma` *set_union_nil* : ∀ *c*, (*set_union eqA nil c*) == *c*.

`Lemma` *set_union_fold_left*: ∀ {*ls l*},
(⋃*i ls* ; *l*) = *set_union eqA* (⋃*i ls*) *l*.

`End` *Sets*.

Set relation axioms. `Instance` *proper_equivlistA_set_union* : ∀ {*A eqA*},
Proper (*equivlistA eq* ==> *equivlistA eq* ==> *equivlistA eq*) (@*set_union A eqA*).

`Instance` *proper_equivlistA_subset* : ∀ {*A*},
Proper (*equivlistA eq* ==> *equivlistA eq* ==> *impl*) (@*subset A*).

`Instance` *proper_subset_union* : ∀ {*A eqA*},
Proper (*subset* ==> *subset* ==> *subset*) (@*set_union A eqA*).

`Instance` *proper_subset_set_add* : ∀ {*A eqA*},
Proper (*eq* ==> *subset* ==> *subset*) (@*set_add A eqA*).

`Instance` *proper_subset_set_diff* : ∀ {*A eqA*},
Proper (*subset* ==> *eq* ==> *subset*) (@*set_diff A eqA*).

`Instance` *proper_equivlistA_set_diff* : ∀ {*A eqA*},
Proper (*equivlistA eq* ==> *eq* ==> *equivlistA eq*) (@*set_diff A eqA*).

`Instance` *proper_equivlistA_fold_union* : ∀ {*A eqA*} (*l* : *list* (`set` *A*)),
Proper (*equivlistA eq* ==> *equivlistA eq*) (*fold_left* (@*set_union A eqA*) *l*).

B.3.2 Sequences

Sequences are naturally encoded as cons-lists as well, and we do not hide the constructors as we did for sets. We prove a couple of extra lemmas to complement the ones in Coq's standard library. These will be useful when mixing list indexing and list membership.

`Lemma` *nth_error_Forall_inv* :
∀ {*A*} {*x* : *A*} {*xs*} {*i*} *P*,
nth_error xs i = *Some x* → *Forall P xs* → *P x*.

`Lemma` *nth_error_in* : ∀ {*A*} (*x* : *A*) *xs i*, *nth_error xs i* = *Some x* → *In x xs*.

`Definition` *concat* {*A*} (*fs* : *list* (*list A*)) := *fold_left* (@*app A*) *fs nil*.

B.3.3 Streams

Streams are encoded (coinductively) as follows in Coq's standard library:

```
CoInductive Stream : Type :=
    Cons : A → Stream → Stream.
```

We import it qualified.

```
Require Coq.Lists.Streams.
```

```
Lemma ForAll_Cons : ∀ {A} P (v : A) vs, Streams.ForAll P (Streams.Cons v vs) → Streams.ForAll P vs.
```

B.3.4 Partiality

Following the functional programming language tradition of ML and Haskell, we encode partial functions within a total setting through an option data type (coproduct with unit). In order to ease manipulation of these, we define the components of the option functor and monad, to ease manipulation of these values.

```
Definition fmap_option {A B} (f : A → B) (oa : option A) : option B :=
  match oa with
  | Some a ⇒ Some (f a)
  | None ⇒ None
  end.
```

```
Fixpoint lift2_option {A B C} (f : A → B → C) (oa : option A) (ob : option B) : option C :=
  match (oa, ob) with
  | (Some a, Some b) ⇒ Some (f a b)
  | (None, _) ⇒ None
  | (_, None) ⇒ None
  end.
```

```
Definition bind_option {A B} (oa : option A) (k : A → option B) : option B :=
  match oa with
  | Some a ⇒ k a
  | None ⇒ None
  end.
```

```
Definition sequence_option {A} (oxs : list (option A)) : option (list A) :=
  let k m m' := bind_option m (fun x ⇒ bind_option m' (fun xs ⇒ Some (cons x xs)))
  in fold_right k (Some nil) oxs.
```

```
Fixpoint fold_left_option {A B} (f : A → B → option A) (a : A) (bs : list B) : option A :=
  match bs with
  | cons b bs' ⇒ match f a b with
                 | Some a' ⇒ fold_left_option f a' bs'
```

```
                    | None ⇒ None
                    end
  | nil ⇒ None
  end.

Inductive lift2_option_prop {A} (R : relation A) : relation (option A) :=
  | lift2_somes : ∀ x y, R x y → lift2_option_prop R (Some x) (Some y).
```

B.4 Mathematical structures

Coq has excellent support for defining arbitrary equivalence relations and supports rewrites over these equivalence relations when sound. It achieves this by using a mechanism similar to Haskell's type classes. This allows the proof developer to define abstract structures such as partial orders and equivalence relations, as well as the properties that these must obey.

```
Require Export Coq.Classes.RelationClasses.
Require Export Coq.Relations.Relation_Operators.
```

For instance, we prove that sets are substitutable by equal sets within subset statements:

```
Instance proper_equivlist_In : ∀ {A}, Proper (@eq A ==> equivlistA (@eq A) ==> impl) (@In A).

Defined.

Instance proper_equivlist_subset : ∀ {A}, Proper (equivlistA eq ==> equivlistA eq ==> impl) (@subset A).
```

This allows us to use Coq's rewrite tactic to rewrite set-equal lists within subset statements easily.
The need to prove these trivial results is a consequence of the encoding we chose for sets, and allow Coq to be slightly smarter in handling them. Whenever we invoke the rewrite tactic, Coq will search within the instances of Proper and try to determine a set of instances that will allow it to do the rewrite. We define a full set of rewrite tactics for the subset relation and union according to set theory which we omit, and which will apply whenever we use the rewrite tactic.

B.4.1 Tactics and Notations

We will now present some custom tactics and notations that we use in the proofs.
A useful tactic for destroying an iterated conjunction in one go. `Ltac splits := split ; try splits.`

The case tagging technique of Pierce et al [PCG+10].

```
Require Export String.
Open Scope string_scope.
```

```
Tactic Notation "Case_aux" ident(x) constr(name) :=
  first [
    set (x := name); move_to_top x
  | assert_eq x name
  | fail 1 "because we are working on a different case." ].

Ltac Case name := Case_aux case name.
Ltac SCase name := Case_aux subcase name.
Ltac SSCase name := Case_aux subsubcase name.
```

Some lemmas that allow us change the form of goals, by replacing terms with quantified variables and adding equality hypotheses. `Lemma` *iforall* : $\forall \{A\}\ \{P : A \to \mathtt{Prop}\}\ \{x\}, (\forall\ x', x = x' \to P\ x') \to P\ x$.

`Lemma` *iexists3_l* : $\forall \{A\ B\ C\}\ \{P : A \to B \to C \to \mathtt{Prop}\}\ \{x\ y\ z\},$
$(\exists\ x', \exists\ y', \exists\ z', P\ x'\ y'\ z' \wedge x = x' \wedge y = y' \wedge z = z') \to P\ x\ y\ z.$

`Ltac` *iexists3* := `eapply` *iexists3_l*; `do` 3 `eexists`; `split`.

`Lemma` *iexists4_l* : $\forall \{A\ B\ C\ D\}\ \{P : A \to B \to C \to D \to \mathtt{Prop}\}\ \{x\ y\ z\ w\},$
$(\exists\ x', \exists\ y', \exists\ z', \exists\ w', P\ x'\ y'\ z'\ w' \wedge x = x' \wedge y = y' \wedge z = z' \wedge w = w') \to P\ x\ y\ z\ w.$

`Ltac` *iexists4* := `eapply` *iexists4_l*; `do` 4 `eexists`; `split`.

`Lemma` *remove_false_disj* : $\forall\ P, P \vee \mathit{False} \to P$.

A tactic that searches for equalities and attempts to rewrite them all and clear them from the goal. `Ltac` *rewrites* :=

```
  match goal with
  | H : ?v ?w = ?x ⊢ _ ⇒ rewrite_all ← H; clear H; try rewrites
  | H : ?v = ?x ⊢ _ ⇒ rewrite_all H; clear H; try rewrites
  end.
```

The following boilerplate lemmas allow me to define a simplification tactic that normalises list constructs, by performing map fusion and normalising the tuple decomposition functions that will be ubiquitous in ott-generated code.

`Lemma` *fun_tuple2_decompose* : $\forall \{A\ B\ C\}\ \{f : A \times B \to C\},$
$f =$ `fun` $xy \Rightarrow$ `let` $(x, y) := xy$ `in`
$(\mathtt{fun}\ x \Rightarrow \mathtt{fun}\ y \Rightarrow f\ (x,y))\ x\ y.$

`Lemma` *fun_tuple3_decompose* : $\forall \{A\ B\ C\ D\}\ \{f : A \times B \times C \to D\},$
$f =$ `fun` $xyz \Rightarrow$ `let` $(r, z) := xyz$ `in` `let` $(x, y) := r$ `in`
$(\mathtt{fun}\ x \Rightarrow \mathtt{fun}\ y \Rightarrow \mathtt{fun}\ z \Rightarrow f\ (x, y, z))\ x\ y\ z.$

`Lemma` *fun_tuple4_decompose* : $\forall \{A\ B\ C\ D\ E\}\ \{f : A \times B \times C \times D \to E\},$
$f =$ `fun` $xyzw \Rightarrow$ `let` $(r, w) := xyzw$ `in` `let` $(r', z) := r$ `in` `let` $(x, y) := r'$ `in`

```
    (fun x ⇒ fun y ⇒ fun z ⇒ fun w ⇒ f (x, y, z, w)) x y z w.

Lemma fun_tuple5_decompose : ∀ {A B C D E F} {f : A × B × C × D × E → F},
  f = fun xyzwa ⇒ let (r, a) := xyzwa in let (r', w) := r in let (r'', z) := r' in let (x, y) := r'' in
      (fun x ⇒ fun y ⇒ fun z ⇒ fun w ⇒ fun a ⇒ f (x, y, z, w, a)) x y z w a.

Lemma fun_tuple6_decompose : ∀ {A B C D E F G} {f : A × B × C × D × E × F → G},
  f = fun xyzwab ⇒ let (r, b) := xyzwab in let (r', a) := r in let (r'', w) := r' in let (r''', z) := r''
                   in let (x, y) := r''' in
      (fun x ⇒ fun y ⇒ fun z ⇒ fun w ⇒ fun a ⇒ fun b ⇒ f (x, y, z, w, a, b)) x y z w a b.

Lemma fun_tuple7_decompose : ∀ {A B C D E F G H} {f : A × B × C × D × E × F × G → H},
  f = fun xyzwabc ⇒ let (xyzwab,c) := xyzwabc in let (r, b) := xyzwab in
                    let (r', a) := r in let (r'', w) := r' in let (r''', z) := r'' in
                    let (x, y) := r''' in
      (fun x ⇒ fun y ⇒ fun z ⇒ fun w ⇒ fun a ⇒ fun b ⇒ fun c ⇒ f (x, y,
        z, w, a, b, c)) x y z w a b c.

Lemma map_fusion : ∀ {A B C} {f : B → C} {g : A → B} {xs : list A},
  map f (map g xs) = map (fun x ⇒ f (g x)) xs.

Lemma Forall_map_fusion : ∀ {A B} {f : B → Prop} {g : A → B} {xs : list A},
  Forall f (map g xs) ↔ Forall (fun x ⇒ f (g x)) xs.

Ltac ssimpl :=
  repeat (try (apply Forall_map_fusion);
          try (rewrite map_fusion);
          try (rewrite fun_tuple7_decompose);
          try (rewrite fun_tuple6_decompose);
          try (rewrite fun_tuple5_decompose);
          try (rewrite fun_tuple4_decompose);
          try (rewrite fun_tuple3_decompose);
          try (rewrite fun_tuple2_decompose);
          simpl).

Tactic Notation "ssimpl" "in" ident(H) :=
  repeat (try (apply Forall_map_fusion in H);
          try (rewrite map_fusion in H);
          try (rewrite fun_tuple7_decompose in H);
          try (rewrite fun_tuple6_decompose in H);
          try (rewrite fun_tuple5_decompose in H);
          try (rewrite fun_tuple4_decompose in H);
          try (rewrite fun_tuple3_decompose in H);
```

```
    try (rewrite fun_tuple2_decompose in H);
    simpl in H).
Notation "'funp' ( x , y ) => e" := (fun xy ⇒ match xy with (x,y) ⇒ e end) (at level 100).
```

B.4.2 Environments

Environments are predicated on the existence of decidable equality for the type of the variable.

```
Module Type EnvSig.
Parameter var : Type.
Parameter eq_var : ∀ x x' : var, { x = x' } + { x ≠ x' }.
End EnvSig.
Module Env (Params : EnvSig).
Import Params.
```

We encode environments as partial functions from the domain of variables to a given type.

```
Definition env A := var → option A.
```

The following definitions allow us to manipulate environments as typical:

```
Definition empty {A} : env A := fun _ ⇒ None.
Definition add {A} (x : var) (t : A) (Γ : env A) : env A :=
  fun x' ⇒ if eq_var x x' then Some t else Γ x'.
Definition elem {A} (x : var) (Γ : env A) : Prop :=
  match (Γ x) with
  | Some t' ⇒ True
  | None ⇒ False
  end.
Definition lookup {A} (x : var) (t : A) (Γ : env A) : Prop :=
  match (Γ x) with
  | Some t' ⇒ t = t'
  | None ⇒ False
  end.
```

We also defined some auxiliary lemmas for the interaction between the environment manipulation functions.

```
Lemma lookup_add : ∀ {A} (Γ : env A) x (u : A),
  lookup x u (add x u Γ).
```

```
Lemma lookup_add_ty : ∀ {A} (Γ : env A) x (u : A) (u' : A),
  lookup x u (add x u' Γ) → u = u'.

Lemma lookup_add_dec : ∀ {A} (Γ : env A) x x' (u u' : A),
  x ≠ x' → lookup x u (add x' u' Γ) → lookup x u Γ.

Lemma lookup_add_dec2 : ∀ {A} (Γ : env A) x x' (u u' : A),
  x ≠ x' → lookup x u Γ → lookup x u (add x' u' Γ).

Lemma comm_add : ∀ {A} x x' (t:A) t' Γ,
  x ≠ x' → add x t (add x' t' Γ) = add x' t' (add x t Γ).

Lemma idem_add : ∀ {A} x (t:A) t' Γ,
  add x t (add x t' Γ) = add x t Γ.

Lemma uniq_lookup : ∀ {A l} {t : A} {t' Δ},
  lookup l t Δ → lookup l t' Δ → t = t'.

End Env.
```

B.5 Auxiliary definitions

In this section we will present the Coq encodings of the auxiliary definitions present in figures 6.4 and 6.6. Note that in some cases we will just assume the existence of a function and state its properties. For more exhaustive guarantees of correctness, we should define them and prove that they obey the properties in question. In most cases though, these are observable in the definitions of the function in the aforementioned figures. In some cases we will have to refer to the term constructors with their Coq names, so we will start by transcribing what ott generated for the grammars:

```
Definition l : Set := nat.

Inductive bl : Set :=
 | bl_BLbl (l5:l).

Inductive t : Set :=
 | TyVar (X:typevar)
 | TyArr (t5:t) (t':t).

Inductive r : Set :=
 | RTExprVar (x:termvar)
 | RTExprLam (x:termvar) (r5:r)
```

```
 | RTExprApp (r5:r) (r':r)
 | RTExprLbl (l5:l)
 | RTExprKCase (l5:l) (_:list (r×r))
 | RTExprCApp (l5:l) (r5:r)
 | RTExprPrim (x:termvar)
 | RTExprRKCase (bl5:bl) (r_5:r) (_:list (r×r)).

Inductive T : Set :=
 | AltSimpTy (t5:t)
 | AltAlt (l5:l) (_:list (r×T)).

Definition nn : Set := nat.

Definition k : Set := nat.

Inductive ts : Set :=
 | SchemeAlt (T5:T)
 | SchemeForall (X:typevar) (ts5:ts).

Definition F : Set := (Streams.Stream typevar).

Definition s : Set := list (typevar × t).

Definition H : Set := (termvar → option ts).

Definition c : Set := (list label).

Definition lm : Set := list (label × Kappa.r).

Definition E : Set := (label → Streams.Stream r).

Definition D : Set := (label → option t).
```

In order to define the inference rules presented throughout the thesis, we then added the following definitions:

```
Parameter Δ : D.

Definition beq_r r r' := if (eq_r r r') then true else false.
```

```
Definition beq_label l l' := if (eq_label l l') then true else false.

Definition pop (Θ : Kappa.E) (l : label) : (Kappa.r × Kappa.E) :=
  match Θ l with
  | (Streams.Cons v vs) ⇒ (v, fun l' ⇒ if eq_label l l' then vs else Θ l')
  end.

Fixpoint subst_r (r_6:Kappa.r) (x5:termvar) (r__7:Kappa.r) {struct r__7} : Kappa.r :=
  match r__7 with
  | (RTExprVar x) ⇒ (if eq_termvar x x5 then r_6 else (RTExprVar x))
  | (RTExprLam x r5) ⇒ RTExprLam x (if eq_termvar x5 x then r5 else subst_r r_6 x5 r5)
  | (RTExprApp r5 r') ⇒ RTExprApp (subst_r r_6 x5 r5) (subst_r r_6 x5 r')
  | (RTExprLbl l) ⇒ RTExprLbl l
  | (RTExprKCase l vrs5) ⇒
      let vrs' := (map (funp (v,r) ⇒ (v, subst_r r_6 x5 r)) vrs5)
      in RTExprKCase l vrs'
  | (RTExprCApp l r5) ⇒ RTExprCApp l (subst_r r_6 x5 r5)
  | (RTExprPrim x) ⇒ RTExprPrim x
  | (RTExprRKCase bl5 e5 vrs5) ⇒
      let vrs' := (map (funp (v,r) ⇒ (v, subst_r r_6 x5 r)) vrs5)
      in RTExprRKCase bl5 e5 vrs'
  end.

Fixpoint subst_l_r (r_6:Kappa.r) (x5:label) (r__7:Kappa.r) {struct r__7} : Kappa.r :=
  match r__7 with
  | (RTExprVar x) ⇒ (RTExprVar x)
  | (RTExprLam x r5) ⇒ RTExprLam x (subst_l_r r_6 x5 r5)
  | (RTExprApp r5 r') ⇒ RTExprApp (subst_l_r r_6 x5 r5) (subst_l_r r_6 x5 r')
  | (RTExprLbl l) ⇒ (if eq_label l x5 then r_6 else (RTExprLbl l))
  | (RTExprKCase l vrs5) ⇒
      let vrs' := (map (funp (v,r) ⇒ (v,subst_l_r r_6 x5 r)) vrs5)
      in (if eq_label l x5 then RTExprRKCase (bl_BLbl l) r_6 vrs' else RTExprKCase l vrs')
  | (RTExprCApp l r5) ⇒
      if eq_label l x5 then (subst_l_r r_6 x5 r5) else RTExprCApp l (subst_l_r r_6 x5 r5)
  | (RTExprPrim x) ⇒ RTExprPrim x
  | (RTExprRKCase bl5 e5 vrs5) ⇒
      let vrs' := (map (funp (v,r) ⇒ (v,subst_l_r r_6 x5 r)) vrs5)
      in RTExprRKCase bl5 e5 vrs'
  end.
```

```
Fixpoint arrT (t : Kappa.t) (T : Kappa.T) : Kappa.T :=
  match T with
  | AltSimpTy t' ⇒ AltSimpTy (TyArr t t')
  | AltAlt l vTs ⇒ AltAlt l (map (funp (v,T') ⇒ (v, arrT t T')) vTs)
  end.

Definition Rep := list (list (label × Kappa.r) × Kappa.t).

Fixpoint rep (T:Kappa.T) : Rep :=
  match T with
  | (AltSimpTy t) ⇒ cons (nil, t) nil
  | (AltAlt l vTsl) ⇒
      (((fix repvTs (vTs:list (Kappa.r×Kappa.T)) : Rep :=
        match vTs with
        | nil ⇒ nil
        | cons (v,T') vTs' ⇒
            app (map (funp (lms, t') ⇒ (cons (l,v) lms, t'))
                     (map (funp (lm', t') ⇒ (rm_all_f (funp (l', _) ⇒ beq_label l' l) lm', t'))
(rep T')))
                 (repvTs vTs')
        end)) vTsl)
  end.

Definition consistent (r : Rep) : Prop :=
  ∀ lm t lm' t', In (lm, t) r → In (lm', t') r → (lm ⊆ lm' → (lm = lm' ∧ t = t')).

Definition leq (T : Kappa.T) (T' : Kappa.T) : Prop := ∀ lm t,
  In (lm, t) (rep T) → ∃ lm', In (lm', t) (rep T') ∧ lm ⊆ lm'.
Notation "T ≤ T'" := (leq T T') (at level 5).

Definition leq' T T' : Prop :=
  (∀ lm' t', In (lm', t') (rep T') → ∃ lm, In (lm, t') (rep T) ∧ lm ⊆ lm').

Definition leq_red (T : Kappa.T) (T' : Kappa.T) : Prop :=
  leq T T' ∧ leq' T T'.

Notation "r ⊑ r'" := ( leq_red r r') (at level 5).
Inductive canonical (T : Kappa.T) (T' : Kappa.T) :=
  canonical_proof : (∀ T'', T'' ⊑ T → T' ⊑ T'') → canonical T T'.

Definition eq_dec (A : Type) := ∀ x x' : A, { x = x' } + { x ≠ x' }.

Lemma prod_eq_dec : ∀ {A B : Type},
  eq_dec A → eq_dec B → (∀ x x' : (A × B), { x = x' } + { x ≠ x' }).
```

See the body of the thesis for a proof of decidability of canonical.

```
Parameter canonical_dec : T → T.

Axiom canonical_dec_canonical : ∀ {T T'},
  canonical_dec T = T' ↔ canonical T T'.

Definition compat (T : Kappa.T) (c : Kappa.c) : Prop := ∀ lm t l v,
  In (lm, t) (rep (canonical_dec T)) ∧ In (l, v) lm → ~ (In l c).

Fixpoint lookup_al {A B} (eq_A : ∀ x x' : A, { x = x' } + { x ≠ x' })
                         (el : A) (xs : list (A × B)) : option B :=
  match xs with
  | nil ⇒ None
  | cons (a,b) xs' ⇒ if eq_A a el then Some b else lookup_al eq_A el xs'
  end.
```

We use this parameter as a function standing for the relation defined by the inference rules in ott.

```
Parameter compute_app : Kappa.T → list (label × Kappa.r) → Kappa.T → option Kappa.T.

Definition appT (T : Kappa.T) (T' : Kappa.T) : option Kappa.T := compute_app T nil T'.
```

Due to the dependencies in ott, this is defined with an axiom (exhaustive_def) later on.

```
Parameter exhaustive : (list (r × r)) → t → Prop.

Definition simT1 (T : Kappa.T) (T' : Kappa.T) : Prop := T ⊑ T' ∨ T' ⊑ T.
Definition simT : relation T := clos_trans T simT1.
Notation "r ~ r'" := (simT r r') (at level 5).

Fixpoint appSt (s : Kappa.s) (t : Kappa.t) :=
  match t with
  | (TyVar X) ⇒ match find (fun (Y,_) ⇒ if eq_typevar X Y then true else false) s with
                | Some (_,t') ⇒ t'
                | None ⇒ t
                end
  | (TyArr t1 t2) ⇒ TyArr (appSt s t1) (appSt s t2)
  end.

Fixpoint appST (s : Kappa.s) (T : Kappa.T) :=
  match T with
  | (AltSimpTy t) ⇒ AltSimpTy (appSt s t)
  | (AltAlt l vTs) ⇒ AltAlt l (map (fun (v,T) ⇒ (v, appST s T)) vTs)
  end.
```

```
Fixpoint appSts (s : Kappa.s) (ts : Kappa.ts) :=
  match ts with
  | (SchemeAlt T) ⇒ SchemeAlt (appST s T)
  | (SchemeForall X ts) ⇒ SchemeForall X (appSts (filter (funp (Y,_) ⇒ if eq_typevar X Y then false
else true) s) ts)
  end.

Definition appSΓ (s : Kappa.s) (Γ : Kappa.H) : Kappa.H :=
  fun x ⇒ match Γ x with
           | Some ts ⇒ Some (appSts s ts)
           | None ⇒ None
           end.

Definition compS (s1 s2 : Kappa.s) :=
  app (map (funp (x,t) ⇒ (x, appSt s1 t)) s2) s1.

Inductive ts_list : Set :=
  | SchemeForallList (Xs: list typevar) (T : Kappa.T).

Fixpoint ts_iso_ts_list (ts : Kappa.ts) : Kappa.ts_list :=
  match ts with
  | SchemeAlt T ⇒ SchemeForallList nil T
  | SchemeForall X ts' ⇒ match ts_iso_ts_list ts' with
                         | SchemeForallList Xs T ⇒ SchemeForallList (cons X Xs) T
                         end
  end.

Definition appSts_list (s : Kappa.s) (tsl : ts_list) :=
  match tsl with
  | SchemeForallList Xs T ⇒ SchemeForallList Xs (appST (filter (funp (Y,_) ⇒ if in_dec eq_typevar
Y Xs then false else true) s) T)
  end.

Fixpoint ftv_t (t : Kappa.t) : list typevar :=
  match t with
  | TyVar X ⇒ cons X nil
  | TyArr t' t'' ⇒ app (ftv_t t') (ftv_t t'')
  end.

Fixpoint ftv_T (T : Kappa.T) : list typevar :=
  match T with
  | AltSimpTy t ⇒ ftv_t t
  | AltAlt l vTs ⇒ concat (map (funp (v,T') ⇒ ftv_T T') vTs)
```

```
  end.

Fixpoint ftv (ts : Kappa.ts) : list typevar :=
  match ts with
  | SchemeAlt T ⇒ ftv_T T
  | SchemeForall X ts' ⇒ set_diff eq_typevar (ftv ts') (cons X nil)
  end.

Definition subtyp (ts : Kappa.ts) (ts' : Kappa.ts) :=
  match (ts_iso_ts_list ts, ts_iso_ts_list ts') with
  | (SchemeForallList Xs T, SchemeForallList Xs' T') ⇒
      ((∃ s, appST s T = T' ∧ ∀ X t, In (X, t) s → In X Xs) ∧ ∀ X, In X Xs' → ~ (In X (ftv ts)))
  end.

Notation "ts  ts'" := ( subtyp ts ts') (at level 5).

Notation "s1  s2" := ( set_union eq_label s1 s2) (at level 100).
Notation "( i ls ; l )" := (fold_left (set_union eq_label) ls l) (at level 5).
Notation "( i ls )" := (fold_left (set_union eq_label) ls nil) (at level 5).

Fixpoint fv (r : Kappa.r) : list label :=
  match r with
  | RTExprVar x ⇒ cons x nil
  | RTExprLam x r' ⇒ set_diff eq_termvar (fv r') (cons x nil)
  | RTExprLbl l ⇒ nil
  | RTExprApp r1 r2 ⇒ (fv r1) ∪ (fv r2)
  | RTExprCApp l r' ⇒ fv r'
  | RTExprKCase l vrs ⇒ (⋃i (map (funp (v,r'') ⇒ fv r'') vrs))
  | RTExprPrim x ⇒ nil
  | RTExprRKCase (bl_BLbl l) r' vrs ⇒ (⋃i (map (funp (v,r'') ⇒ fv r'') vrs))
  end.

Fixpoint fl (r : Kappa.r) : list label :=
  match r with
  | RTExprVar x ⇒ nil
  | RTExprLam x r' ⇒ fl r'
  | RTExprLbl l ⇒ cons l nil
  | RTExprApp r1 r2 ⇒ (fl r1) ∪ (fl r2)
  | RTExprCApp l r' ⇒ set_diff eq_label (fl r') (cons l nil)
  | RTExprKCase l vrs ⇒ (cons l nil) ∪ (⋃i (map (funp (v,r'') ⇒ fl r'') vrs))
  | RTExprPrim x ⇒ nil
  | RTExprRKCase (bl_BLbl l) r' vrs ⇒ (⋃i (map (funp (v,r'') ⇒ fl r'') vrs))
```

```
  end.
```

We only use the properties of inflate that have been justified in the body of the thesis, and assume its existence. Again for more exhaustive guarantees of correctness, this should be fully defined, and we should prove that all the properties still hold.

```
Parameter inflate : T → T → T.

Parameter unifyS : t → t → option s.

Inductive unify_mgu_r (t : Kappa.t) (t' : Kappa.t) (s : Kappa.s) (r : Kappa.s) : Prop :=
| unify_def : (appSt r t = appSt r t' ∧ ∃ s', r = compS s' s) → unify_mgu_r t t' s r.

Axiom unify_mgu_ex : ∀ t t' r, appSt r t = appSt r t' → ∃ mgu, unify_mgu_r t t' mgu r.

Axiom unifyS_def : ∀ {r s t t'}, unify_mgu_r t t' s r ↔ Some s = unifyS t t'.
```

We assume the existence of the unification algorithm.

```
Parameter unify : Kappa.T → Kappa.T → option s.
```

Once again, for compute_appU we assume the existence of a function, which will correspond to the "relation" defined in ott using inference rules. The proof that is missing here is of determinacy, which is obvious from the structural definition.

```
Parameter compute_appU : Kappa.F → Kappa.T → list (label × Kappa.r) → Kappa.T → option
(Kappa.F × Kappa.T × Kappa.s).

Definition appTU (F : Kappa.F) (T : Kappa.T) (T' : Kappa.T) (lm : list (label × Kappa.r)) : option
(Kappa.F × Kappa.T × Kappa.s) :=

    (compute_appU F (canonical_dec T) lm (inflate (canonical_dec T) (canonical_dec T'))).

Fixpoint inst (F : Kappa.F) (ts : Kappa.ts) :=
  match (F, ts) with
  | (F, SchemeAlt T) ⇒ (F, T)
  | (Streams.Cons Y F', SchemeForall X ts') ⇒
      match (inst F' ts') with
      | (Ff, T) ⇒ (Ff, subst_t_T (TyVar Y) X T)
      end
  end.

Definition gen (Γ : Kappa.H) (T : Kappa.T) :=
  (fix genFtvTs ftvTs :=
    (match ftvTs with
```

```
    | cons X Xs ⇒ match (Γ X) with
                  | Some t ⇒ genFtvTs Xs
                  | None ⇒ SchemeForall X (genFtvTs Xs)
                  end
    | nil ⇒ SchemeAlt T
    end)) (ftv (SchemeAlt T)).

Definition replace_T (rrTcs : list (r × r × Kappa.T × c)) (f : r → r → Kappa.T → c → Kappa.T) :=
  map (fun tmp_ : r × r × T × c ⇒
       let (p, c_) := tmp_ in
       let (p0, T_) := p in
       let (v_, r_) := p0 in (((v_, r_), f v_ r_ T_ c_), c_)) rrTcs.
```

B.6 Other auxiliary definitions

In order to keep the presentation concise, and in order to make it match the English proof, we have added other definitions, lemmas and axioms to the environment, which we will omit here. Nevertheless, the proof commentary will describe what those conjectures are for, and argue for their validity. We present here the types for the lemmas and definitions that will appear in the proofs in section B.8 for reference.

```
Lemma arrT_impl_TyArr : ∀ {lm T t t'},
  In (lm,t) (rep (arrT t' T)) ↔ ∃ t'', t = TyArr t' t'' ∧ In (lm, t'') (rep T).

Instance proper_leq_red_arrT : ∀ t, Proper (leq_red ==> leq_red) (arrT t).


Qed.

Axiom exhaustive_def : ∀ {vs t},
  exhaustive vs t → (∀ v, ValTyp empty v t → ∃ i, ∃ r, nth_error vs i = Some (v, r)).

Section r_rect_alt.

Variable
  (P_r : r → Prop).

Definition P_rr (rr : (r×r)) := match rr with (r,r') ⇒ P_r r' end.

Lemma P_r_implies_P_rr : ∀ {r r'}, P_r r → P_rr (r', r).

Hypothesis
  (H_RTExprVar : ∀ (x:termvar), P_r (RTExprVar x))
  (H_RTExprLam : ∀ (x:termvar), ∀ (r5:r), P_r r5 → P_r (RTExprLam x r5))
  (H_RTExprApp : ∀ (r5:r), P_r r5 → ∀ (r':r), P_r r' → P_r (RTExprApp r5 r'))
```

```
  (H_RTExprLbl : ∀ (l5:l), P_r (RTExprLbl l5))
  (H_RTExprKCase : ∀ (r_r'_list:list (r×r)), Forall P_rr r_r'_list → ∀ (l5:l), P_r (RTExprKCase l5
r_r'_list))
  (H_RTExprCApp : ∀ (l5:l), ∀ (r5:r), P_r r5 → P_r (RTExprCApp l5 r5))
  (H_RTExprPrim : ∀ (x:termvar), P_r (RTExprPrim x))
  (H_RTExprRKCase : ∀ (r_r'_list:list (r×r)), Forall P_rr r_r'_list → ∀ (bl5:bl), ∀ (r_5:r), P_r r_5
→ P_r (RTExprRKCase bl5 r_5 r_r'_list)).

Fixpoint r_alt_ind (n:r) : P_r n :=
  match n as x return P_r x with
  | (RTExprVar x) ⇒ H_RTExprVar x
  | (RTExprLam x r5) ⇒ H_RTExprLam x r5 (r_alt_ind r5)
  | (RTExprApp r5 r') ⇒ H_RTExprApp r5 (r_alt_ind r5) r' (r_alt_ind r')
  | (RTExprLbl l5) ⇒ H_RTExprLbl l5
  | (RTExprKCase l5 r_r'_list) ⇒ H_RTExprKCase r_r'_list ((fix r_r'_list_ott_ind (r_r_l:list (r×r))
: Forall P_rr r_r_l :=
      match r_r_l as x return Forall P_rr x with
      | nil ⇒ Forall_nil P_rr
      | cons (r2,r3) xl ⇒ Forall_cons (r2,r3) (P_r_implies_P_rr (r_alt_ind r3)) (r_r'_list_ott_ind
xl)
      end) r_r'_list) l5
  | (RTExprCApp l5 r5) ⇒ H_RTExprCApp l5 r5 (r_alt_ind r5)
  | (RTExprPrim x) ⇒ H_RTExprPrim x
  | (RTExprRKCase bl5 r_5 r_r'_list) ⇒ H_RTExprRKCase r_r'_list ((fix r_r'_list_ott_ind (r_r_l:list
(r×r)) : Forall P_rr r_r_l :=
      match r_r_l as x return Forall P_rr x with
      | nil ⇒ Forall_nil P_rr
      | cons (r2,r3) xl ⇒ Forall_cons (r2,r3) (P_r_implies_P_rr (r_alt_ind r3)) (r_r'_list_ott_ind
xl)
      end) r_r'_list) bl5 r_5 (r_alt_ind r_5)
end.

End r_rect_alt.

Section T_rect_alt.

Variable
  (P_T : T → Prop).

Definition P_rT (rT : (r×T)) := match rT with (r,T) ⇒ P_T T end.

Lemma P_T_implies_P_rT : ∀ {r T}, P_T T → P_rT (r,T).
```

```
Hypothesis
  (H_AltSimpTy : ∀ (t_ : t), P_T (AltSimpTy t_))
  (H_AltAlt : ∀ (vTs : list (r×T)), Forall P_rT vTs → ∀ (l_:l), P_T (AltAlt l_ vTs)).

Fixpoint T_alt_ind (T_:T) : P_T T_ :=
  match T_ as x return P_T x with
  | (AltSimpTy t_) ⇒ H_AltSimpTy t_
  | (AltAlt l_ vTs) ⇒ H_AltAlt vTs ((fix vTs_ind (vTs' : list (r×T)) : Forall P_rT vTs' :=
      match vTs' as x return Forall P_rT x with
      | nil ⇒ Forall_nil P_rT
      | cons (v_',T_') vTs'' ⇒ Forall_cons (v_',T_') (P_T_implies_P_rT (T_alt_ind T_')) (vTs_ind
vTs'')
      end) vTs) l_
  end.

End T_rect_alt.

Lemma addΓ_typ_alg1 : ∀ {Γ e x t T c},
  Γ x = None → Γ ⊢a e : T ! c → (add x t Γ) ⊢a e : T ! c.

Axiom Γ_ind : ∀ P, P empty → (∀ (x : termvar) (ts : Kappa.ts) (Γ : H), Γ x = None → P Γ → P (add
x ts Γ)) → ∀ Γ, P Γ.

Lemma addΓ_typ_alg : ∀ {Γ e T c},
  empty ⊢a e : T ! c → Γ ⊢a e : T ! c.
Lemma ci_in_union : ∀ {v r T c v_r_T_c ci},
  c == (fold_left (set_union eq_label)
            (map
              (fun pat_ : Kappa.r × Kappa.r × Kappa.T × Kappa.c ⇒
                let (p, c_) := pat_ in
                let (p0, _) := p in let (_, _) := p0 in c_) v_r_T_c) nil)
  → In (((v, r), T), ci) v_r_T_c
  → ci ⊆ c.

Lemma appS_dist_add : ∀ {s x t Γ},
  (appSΓ s (add x (SchemeAlt (AltSimpTy t)) Γ)) = (add x (SchemeAlt (AltSimpTy (appSt s t)))
(appSΓ s Γ)).

Lemma appSt_nil_id : ∀ {t},
  appSt nil t = t.

Lemma appST_nil_id : ∀ {T},
  appST nil T = T.

Lemma appSts_nil_id : ∀ {ts},
```

appSts nil ts = ts.

Lemma *appSΓ_nil_id* : ∀ {*x ts Γ*},
lookup x ts Γ → *lookup x ts* (*appSΓ nil Γ*).

Lemma *filter_const_true* : ∀ {*A f*} {*xs* : *list A*},
(∀ *x*, *f x* = *true*) → *filter f xs* = *xs*.

Lemma *prec_determines_altty* : ∀ {*T T'*},
(*SchemeAlt T*) ≤ (*SchemeAlt T'*) → *T* = *T'*.

Lemma *prec_determines_altty2* : ∀ {*ts T*},
(*SchemeAlt T*) ≤ *ts* → ∃ *Xs*, *ts_iso_ts_list ts* = *SchemeForallList Xs T*.

Definition *ftv_ts_list* (*tsl* : *ts_list*) : *list typevar* :=
match *tsl* with
| *SchemeForallList Xs T* ⇒ *set_diff eq_typevar* (*ftv_T T*) *Xs*
end.

Lemma *ftv_ftv_ts_list* : ∀ {*ts*},
(*ftv ts*) == (*ftv_ts_list* (*ts_iso_ts_list ts*)).

Instance *reflexivity_prec* : *Reflexive subtyp*.

Lemma *filter_snd_comp* : ∀ {*A B p1 p2*} {*l* : *list* (*A* × *B*)},
filter (fun *ab* : *A* × *B* ⇒ let (*a*,_) := *ab* in *p1 a*) (*filter* (fun *ab* : *A* × *B* ⇒ let (*a*,_) := *ab* in (*p2 a*)) *l*)
= *filter* (fun *ab* : *A* × *B* ⇒ let (*a*,_) := *ab* in *andb* (*p1 a*) (*p2 a*)) *l*.

Lemma *not_in_dec_cons* : ∀ {*A Xs eqA*} {*X Y* : *A*},
(if *in_dec eqA Y* (*X* :: *Xs*) then *false* else *true*) = *andb* (if *in_dec eqA Y Xs* then *false* else *true*) (if *eqA X Y* then *false* else *true*).

Lemma *appSts_appSts_list* : ∀ {*s ts*},
ts_iso_ts_list (*appSts s ts*) = *appSts_list s* (*ts_iso_ts_list ts*).

Lemma *appST_arrT* : ∀ {*s t T*},
appST s (*arrT t T*) = *arrT* (*appSt s t*) (*appST s T*).

Axiom *Δ_no_ftv* : ∀ *l t*, *lookup l t Δ* → *ftv* (*SchemeAlt* (*AltSimpTy t*)) = *nil*.

Fixpoint *sfirst* {*A*} (*n* : *nat*) (*s* : *Streams.Stream A*) : *list A* :=
match (*n*, *s*) with
| (*O*, _) ⇒ *nil*
| (*S m*, *Streams.Cons x s'*) ⇒ *cons x* (*sfirst m s'*)
end.

Fixpoint *rm_subst* (*X* : *typevar*) (*s* : *Kappa.s*) :=

```
  match s with
  | cons (Y, t) s' ⇒ if eq_typevar X Y then rm_subst X s' else cons (Y, t) (rm_subst X s')
  | nil ⇒ nil
  end.

Definition rm_substs (s : Kappa.s) (tvs : list typevar) := fold_left compose (map rm_subst tvs) id s.

Lemma find_app : ∀ {A} {p} {l1 l2 : list A},
  find p (l1 ++ l2) = match find p l1 with
                      | Some x ⇒ Some x
                      | None ⇒ find p l2
                      end.

Lemma find_fst_map_snd_fusion : ∀ {A B} {p} {f : B → B} {l : list (A × B)},
  find (funp (a,b) ⇒ p a) (map (funp (a,b) ⇒ (a, f b)) l) = fmap_option (funp (a, b) ⇒ (a, f b)) (find
(funp (a,b) ⇒ p a) l).

Lemma filter_app : ∀ {A} {p} {l1 l2 : list A},
  filter p (l1 ++ l2) = app (filter p l1) (filter p l2).

Lemma filter_fst_map_snd_flip : ∀ {A B} {p} {f : B → B} {l : list (A × B)},
  filter (funp (a,b) ⇒ p a) (map (funp (a,b) ⇒ (a, f b)) l) = map (funp (a,b) ⇒ (a, f b)) (filter (funp
(a,b) ⇒ p a) l).

Lemma appSt_compS : ∀ s1 s2 t,
  appSt s1 (appSt s2 t) = appSt (compS s1 s2) t.

Lemma appST_compS : ∀ {s1 s2 T},
  appST s1 (appST s2 T) = appST (compS s1 s2) T.

Lemma take_drop_app_fusion : ∀ {A k} {xs : list A}, app (take k xs) (drop k xs) = xs.
Lemma take_drop_take_app_fusion : ∀ {A k k'} {xs : list A},
  app (take k xs) (take k' (drop k xs)) = (take (k + k') xs).
Lemma length_map_fusion : ∀ {A B} {xs : list A} {f : A → B}, List.length (map f xs) = List.length xs.
Lemma appST_AltSimpTy : ∀ {s t},
  appST s (AltSimpTy t) = (AltSimpTy (appSt s t)).
Lemma appST_AltAlt_inv : ∀ {s T l vTs},
  appST s T = AltAlt l vTs → ∃ vTs', T = AltAlt l vTs'.
Axiom compute_app_CA : ∀ {T T' T'' lm},
  Some T'' = compute_app T lm T' ↔ CA T lm T' T''.

Axiom compute_appU_CAU : ∀ {F F' T T' T'' lm s},
  Some (F',T'',s) = compute_appU F T lm T' ↔ CAU T lm T' s T''.

Lemma lm_nil_simpTy : ∀ {t T},
```

In (*nil*, *t*) (*rep T*) → *T* = (*AltSimpTy t*).

`Lemma` *AltAlt_lm_not_nil* : ∀ {*l vTs t*},
~(*In* (*nil*, *t*) (*rep* (*AltAlt l vTs*))).

`Lemma` *lookup_al_in* : ∀ {*A B eqA l*} {*el* : *A*} {*el2* : *B*},
Some el2 = lookup_al eqA el l → *In* (*el*, *el2*) *l*.

`Lemma` *appT_AltSimpTy* : ∀ {*T t Tf*},
Some Tf = appT (*AltSimpTy t*) *T* → ∃ *t'*, *T* = (*AltSimpTy t'*).
`Instance` *proper_arrT_simT* : ∀ *t*, *Proper* (*simT* ==> *simT*) (*arrT t*).

`Qed`.

`Lemma` *Streams_tl_cons* : ∀ {*A*} {*n s*} {*X* : *A*} {*s'*},
Streams.Str_nth_tl n s = Streams.Cons X s' → ∃ *n'*, *Streams.Str_nth_tl n' s = s'*.
Introduction patterns for induction on ⊢ and $\vdash_a$ derivations. The names should be fairly explanatory. In Coq, all the universally quantified variables need to be explictly introduced first, followed by the hypotheses. Induction hypotheses immediately follow the hypothesis that gave rise to them. Moreover, list forms are encoded in coq as lists of tuples of all the indexed variables. This appears in the introduction patterns for the (A)Cases/(A)LCases rules as *rrTcs* which will have type `list (r * r * T * c)`. `Ltac` *induction_KTyp H* :=
`induction` *H* `as`
[*Γ' x' ts' Hts_ts' Hlookup*
| *Γ' x' r' t' T' c' HT_T' HKTyp IHKTyp*
| *Γ' l' t' Hlookup*
| *Γ' l' r' T' c' t' HT_T' HKTyp IHKTyp Hlookup*
| *Γ' r1 r2 T' c1 c2 T1 T2 HT_T' HT_T1 HT_T2 HKTyp1 IHKTyp1 HKTyp2 IHKTyp2*
| *rrTcs Γ' l' c' t'*
Hv_rrTcs HT_rrTcs Hlookup HKTyp_rrTcs1_v HKTyp_rrTcs1 HKTyp_rrTcs2 Hunion Hexhaust
| *rrTcs Γ' l' r' c' t'*
Hv_rrTcs HT_rrTcs Hlookup HKTyp_rrTcs1_v HKTyp_rrTcs1 HKTyp_rrTcs2 HKTyp Hunion Hexhaust
| *Γ' r' T' c' T'' HT_T HT_T' HKTyp IHKTyp HsimT*
| *Γ' r' ts' cs' ts'' Hts_ts' Hts_ts'' HKTyp IHKTyp Hsubtyp*
| *Γ' r' X ts' cs' Hts_ts' HKTyp IHKTyp HnotInEnv*
].
`Ltac` *induction_KTypA H* :=

```
induction H as
  [ Γ' x' ts' ts'' Hts_ts'' Hlookup Hsubtyp
  | Γ' x' r' t' T' c' HT_T' HKTyp IHKTyp
  | Γ' l' t' Hlookup
  | Γ' l' r' T' c' t' HT_T' HKTyp IHKTyp Hlookup
  | Γ' r1 r2 T' c1 c2 T1 T2 T1' T2'
      HT_T' HT_T1 HT_T2 HT_T1' HT_T2'
      HTyp1 IHKTyp1 HTyp2 IHKTyp2 HappT HsimT1 HsimT2 Hcompat1 Hcompat2
  | rrTcs Γ' l' c' t'
      Hv_rrTcs HT_rrTcs Hlookup HKTyp_rrTcs1_v HKTyp_rrTcs1 HKTyp_rrTcs2 Hunion Hexhaust
  | rrTcs Γ' l' r' c' t'
      Hv_rrTcs HT_rrTcs Hlookup HKTyp_rrTcs1_v HKTyp_rrTcs1 HKTyp_rrTcs2 HKTyp Hunion Hexhaust
  ].
```

Statements of independence of parts of the unified list of tuples created by ott. `Lemma` *replace_T_only_T* : ∀ {*f rrTcs v_ r_ T_ c_*},
In (((*v_*, *r_*), *T_*), *c_*) (*replace_T rrTcs f*) → ∃ *T_'* : *Kappa.T*, *In* (((*v_*, *r_*), *T_'*), *c_*) *rrTcs*.

```
Lemma independence_rrTcs_ACases : ∀ {Γ l5 rrTcs c'} f,
  (Γ ⊢a (RTExprKCase l5
              (map
                 (fun pat_ : r × r × Kappa.T × c ⇒
                   let (p, _) := pat_ in
                   let (p0, _) := p in let (v_, r_) := p0 in v_  r_) (replace_T rrTcs f)))
        : AltAlt l5
              (map
                 (fun pat_ : r × r × Kappa.T × c ⇒
                   let (p, _) := pat_ in
                   let (p0, T_) := p in let (v_, _) := p0 in v_  T_) (replace_T rrTcs f))
        ! c')
  → (Γ ⊢a (RTExprKCase l5
              (map
                 (fun pat_ : r × r × Kappa.T × c ⇒
                   let (p, _) := pat_ in
                   let (p0, _) := p in let (v_, r_) := p0 in v_  r_) rrTcs))
        : AltAlt l5
              (map
```

```
                (fun pat_ : r × r × Kappa.T × c ⇒
                 let (p, _) := pat_ in
                 let (p0, T_) := p in let (v_, _) := p0 in v_  T_) (replace_T rrTcs f))
        ! c').

Lemma independence_rrTcs_ALCases : ∀ {Γ l5 v5 rrTcs c'} f,
  (Γ ⊢a (RTExprRKCase (bl_BLbl l5) v5
            (map
                (fun pat_ : r × r × Kappa.T × c ⇒
                 let (p, _) := pat_ in
                 let (p0, _) := p in let (v_, r_) := p0 in v_  r_) (replace_T rrTcs f)))
        : AltAlt l5
            (map
                (fun pat_ : r × r × Kappa.T × c ⇒
                 let (p, _) := pat_ in
                 let (p0, T_) := p in let (v_, _) := p0 in v_  T_) (replace_T rrTcs f))
        ! c')
  → (Γ ⊢a (RTExprRKCase (bl_BLbl l5) v5
            (map
                (fun pat_ : r × r × Kappa.T × c ⇒
                 let (p, _) := pat_ in
                 let (p0, _) := p in let (v_, r_) := p0 in v_  r_) rrTcs))
        : AltAlt l5
            (map
                (fun pat_ : r × r × Kappa.T × c ⇒
                 let (p, _) := pat_ in
                 let (p0, T_) := p in let (v_, _) := p0 in v_  T_) (replace_T rrTcs f))
        ! c').

Lemma lookup_not_in_fail : ∀ {A B eq_dec} {x : A} {l},
  (∀ (y : B), ~ In (x, y) l) → lookup_al eq_dec x l = None.

Lemma weakening_alg : ∀ {x Γ r T c ts},
  Γ x = None → Γ ⊢a r : T ! c → (add x ts Γ) ⊢a r : T ! c.

Instance proper_leq_arrT : ∀ t, Proper (leq ==> leq) (arrT t).


Instance preorder_subset : ∀ A, PreOrder (@subset A).

Instance order_subset : ∀ A, PartialOrder (equivlistA eq) (@subset A).

Instance proper_consistent_equivlist : Proper (equivlistA eq ==> impl) consistent.
```

```
Notation "T =  T'" := ( equivRep T T') (at level 5).

Instance proper_leq_equivrep : Proper (equivRep ==> equivRep ==> impl) leq.

Instance preorder_subtyp : PreOrder subtyp.

Instance preorder_leq : PreOrder leq.

Instance preorder_leq' : PreOrder leq'.


Qed.

Instance equivalence_equivRep : Equivalence equivRep.

Instance order_leq : PartialOrder equivRep leq.


Definition flip_leq_red : ∀ {T T'},
  (∀ lm t, In (lm, t) (rep T)
      → ∀ lm' t', In (lm', t') (rep T') → lm ⊆ lm' → t = t')
  →
  (∀ lm' t', In (lm', t') (rep T')
      → ∀ lm t, In (lm, t) (rep T) → lm ⊆ lm' → t = t').

Instance preorder_leq_red : PreOrder leq_red.

Instance order_leq_red : PartialOrder equivRep leq_red.

Example leq_ex1 : ∀ t1 t2 l v1 v2,
  (leq (AltSimpTy t1) (AltAlt l (cons (v1, (AltSimpTy t1)) (cons (v2, (AltSimpTy t2)) nil)))).

Lemma leq_simpty : ∀ t t', leq (AltSimpTy t) (AltSimpTy t') → t = t'.

Lemma leq_red_simpty : ∀ t t', (AltSimpTy t) ⊑ (AltSimpTy t') → t = t'.

Lemma leq_red_T_simpTy : ∀ T t, T ⊑ (AltSimpTy t) → T = AltSimpTy t.

Instance equiv_simT : Equivalence simT.

Instance proper_subset_compat : ∀ T, Proper (flip subset ==> impl) (compat T).

Lemma AltSimpTy_simT1 : ∀ {t_ T_},
  (AltSimpTy t_) ⊑ T_ ∨ T_ ⊑ (AltSimpTy t_) → (AltSimpTy t_) ⊑ T_.

Lemma AltAlt_not_nil: ∀ {l vTs}, ∀ t, ~ In (nil, t) (rep (AltAlt l vTs)).

Lemma canonical_AltSimpTy2 : ∀ {t},
  canonical_dec (AltSimpTy t) = (AltSimpTy t).
Lemma canonical_simT : ∀ {T}, (canonical_dec T) ~ T.
```

```
Require Import Common.Common.
Require Import Kappa.Kappa.
Require Import Kappa.Notation.
Require Import Kappa.Util.
Import kappa_term.

Axiom all_T_are_T : ∀ T, Is_true (is_T_of_T T).
Axiom all_ts_are_ts : ∀ ts, Is_true (is_ts_of_ts ts).

Lemma subst_typ : ∀ {Γ e T c} s,
  Γ  a e : T ! c → (appSΓ s Γ)  a e : (appST s T) ! c.
Lemma appSts_compS : ∀ {s1 s2 ts},
  appSts s1 (appSts s2 ts) = appSts (compS s1 s2) ts.

Lemma appSΓ_compS : ∀ {s1 s2 Γ},
  appSΓ s1 (appSΓ s2 Γ) = appSΓ (compS s1 s2) Γ.
Lemma appSΓ_fold_compS : ∀ {ss1 ss2 Γ},
  (appSΓ (fold_left compS ss1 nil) (appSΓ (fold_left compS ss2 nil) Γ)) = appSΓ (fold_left compS (app
ss1 ss2) nil) Γ.

Lemma appST_fold_compS : ∀ {ss1 ss2 T},
  (appST (fold_left compS ss1 nil) (appST (fold_left compS ss2 nil) T)) = appST (fold_left compS
(app ss1 ss2) nil) T.

Lemma In_exists_list : ∀ {Q P rrTcs},
      (∀ v_ r_ T_ c_, In (((v_, r_), T_), c_) rrTcs → Q → ∃ T_', ∃ Tr_' : Kappa.T, P r_ T_ T_' Tr_'
c_)
  → (∃f, ∀ v_ r_ T_ T_' Tr_' c_, In (((v_, r_), T_), c_) rrTcs → Q → In (((v_, r_), T_'), c_) (replace_T
rrTcs f) ∧ P r_ T_ T_' Tr_' c_).

Lemma rep_consistent : ∀ T, consistent (rep T).

Definition equivRep (T:Kappa.T) (T':Kappa.T) : Prop :=
  (∀ lm1 t1, In (lm1, t1) (rep T) →
    ∃ lm2, In (lm2, t1) (rep T') ∧ lm1 == lm2)
  ∧ (∀ lm1 t1, In (lm1, t1) (rep T') →
        ∃ lm2, In (lm2, t1) (rep T) ∧ lm1 == lm2).

Instance proper_rep_equivrep : Proper (equivRep ==> equivlistA eq) rep.

Lemma appT_arrT : ∀ {t T T' Tf},
  Some Tf = appT (arrT t T) T' → Tf = T ∧ T' ¬ (AltSimpTy t).

Lemma almost_proper_simT_leq : ∀ {T T' T''}, T ¬ T' → T'' ≤ T → ∃ Tf, Tf ≤ T' ∧ T'' ¬ Tf.

Instance proper_simT_appT : Proper (simT ==> simT ==> lift2_option_prop simT) appT.
```

```
Lemma compat_determines_labels : ∀ {T1' T2 T2' T3 c1 Tf},
      Some Tf = appT T1' T2'
  → compat T2' c1
  → T2' ¬ T2
  → T3 ≤ T2
  → ∃ T3', ∃ Tf', T3' ¬ T3 ∧ Some Tf' = appT T1' T3' ∧ compat T3' c1 ∧ Tf' ≤ Tf.

Lemma compat_determines_labels2 : ∀ {T1 T1' T2' T3 c2 Tf},
      Some Tf = appT T1' T2'
  → compat T1' c2
  → T1' ¬ T1
  → T3 ≤ T1
  → ∃ T3', ∃ Tf', T3' ¬ T3 ∧ Some Tf' = appT T3' T2' ∧ compat T3' c2 ∧ Tf' ≤ Tf.

Lemma small_app_ex : ∀ {T T1 T2 Tf},
  Some T = appT T1 T2 → T ¬ Tf → ∃ T1', ∃ T2', T1' ¬ T1 ∧ T2' ¬ T2 ∧ Some Tf = appT T1' T2'.

Lemma appT_preserve_simT : ∀ {T1 T1' T2 T2' Tf Tf'},
        T1 ¬ T1'
    → T2 ¬ T2'
    → Some Tf = appT T1 T2
    → Some Tf' = appT T1' T2'
    → Tf ¬ Tf'.

Definition rrTcs_to_AltAlt l rrTcs := (AltAlt l (map
  (fun pat_ : r × r × T × c ⇒
   let (p, _) := pat_ in
   let (p0, T_0) := p in let (v_, _) := p0 in v_   T_0) rrTcs)).

Lemma simT_AltAlt : ∀ {l rrTcs f},
      (∀ v_ r_ T_ c_, In (((v_, r_), T_), c_) rrTcs → (f v_ r_ T_ c_) ¬ T_)
  → (rrTcs_to_AltAlt l rrTcs) ¬ (rrTcs_to_AltAlt l (replace_T rrTcs f)).

Lemma canonical_AltSimpTy : ∀ {t T},
  (AltSimpTy t) ¬ T → canonical_dec T = (AltSimpTy t).

Lemma gen_SchemeAlt_prec : ∀ {Γ T}, (gen Γ T)  ( SchemeAlt T).

Lemma gen_SchemeAlt_prec_eq : ∀ {Γ T T'}, (gen Γ T)  ( SchemeAlt T') → T = T'.

Lemma gen_inst_prec : ∀ {F Γ ts}, (gen Γ (snd (inst F ts)))   ts.

Lemma inst_prec2 : ∀ {F ts}, ts   (SchemeAlt (snd (inst F ts))).

Lemma Forall_prec_gen : ∀ {Γ T ts X},
  (gen Γ T)   ts → ¬ elem X Γ → (gen Γ T)  ( SchemeForall X ts).
```

Lemma *inst_prec* : ∀ {*F F′ T ts*}, (*F′*, *T*) = *inst F ts* → *ts* (*SchemeAlt T*).

Lemma *inflate_simT* : ∀ {*T1 T2*}, (*inflate T1 T2*) ¬ *T2*.

Lemma *appST_simT* : ∀ {*s T T′*}, *T* ¬ *T′* → (*appST s T*) ¬ (*appST s T′*).

Lemma *simT_AltSimpTy* : ∀ {*t T*}, (*AltSimpTy t*) ¬ *T* → *leq* (*AltSimpTy t*) *T*.

Lemma *arrT_simT_determines* : ∀ {*t T T′*}, (*arrT t T*) ¬ *T′* → ∃ *T′′*, *T′* = *arrT t T′′*.

Lemma *arrT_simT_pairwise* : ∀ {*t T T′*}, (*arrT t T*) ¬ (*arrT t T′*) → *T* ¬ *T′*.

Lemma *subst_subtyp* : ∀ {*ts ts′*} *s*,
ts *ts′* → (*appSts s ts*) (*appSts s ts′*).

Lemma *canonical_appST_AltSimpTy* : ∀ {*T t s*},
(*canonical_dec* (*appST s T*)) = (*AltSimpTy t*) → ∃ *t′*, *canonical_dec T* = (*AltSimpTy t′*) ∧ *t* = *appSt s t′*.

Lemma *inflate_AltSimpTy* : ∀ {*t t′*},
(*inflate* (*AltSimpTy t*) (*AltSimpTy t′*)) = *AltSimpTy t′*.

Lemma *inflate_appST* : ∀ {*s T1 T2*}, (*appST s* (*inflate T1 T2*)) = *inflate T1* (*appST s T2*).

Lemma *appST_compat* : ∀ {*s T c*}, *compat T c* ↔ *compat* (*appST s T*) *c*.

Lemma *compat_canonical* : ∀ {*T c*}, *compat T c* → *compat* (*canonical_dec T*) *c*.

Lemma *compat_inflate* : ∀ {*T1 T2 c*}, *compat T2 c* → *compat* (*inflate T1 T2*) *c*.

Lemma *compat_arrT* : ∀ {*t T c*}, *compat* (*arrT t T*) *c* → *compat T c*.

Lemma *compat_appT* : ∀ {*T1 T2 Tf c*},
Some Tf = *appT T1 T2* → *compat Tf c* → *compat T1 c* ∧ *compat T2 c*.

Lemma *compat_union* : ∀ {*T c1 c2*},
compat T c1 → *compat T c2* → *compat T* (*c1* *c2*).

Instance *proper_compat_simT* : *Proper* (*simT* ==> *eq* ==> *impl*) *compat*.

Lemma *simT_arrT_simpl* : ∀ {*T1 t T1′ T2 T*},
T1 ¬ (*arrT t T1′*) → *Some T* = *appT T1 T2* → *T1′* ¬ *T* ∧ (*AltSimpTy t*) ¬ *T2*.

Axiom *leq_case* : ∀ {*Ti v v_T*} *l*, *In* (*v*,*Ti*) *v_T* → *Ti* ≤ (*AltAlt l v_T*).

Lemma *lookup_rep_appST* : ∀ {*s T lm*},
lookup_al (*list_eq_dec* (*prod_eq_dec eq_label eq_r*)) *lm* (*rep* (*appST s T*))
= *fmap_option* (*appSt s*) (*lookup_al* (*list_eq_dec* (*prod_eq_dec eq_label eq_r*)) *lm* (*rep T*)).

Lemma *compute_app_subst* : ∀ {*T T′ T′′ lm*} *s*,
Some T′′ = *compute_app T lm T′*
→ *Some* (*appST s T′′*) = *compute_app* (*appST s T*) *lm* (*appST s T′*).

Theorem *fl_preservation*: ∀ Θ Θ′ *e e′*,
  wf Θ → «*e*, Θ –>κ *e′*, Θ′» → (*fl e′*) ⊆ (*fl e*).
Lemma *compute_appU_soundness* : ∀ {*F F′ T1 T2 T s lm*},
    Some (*F′*, *T*, *s*) = *compute_appU F T1 lm T2*
  → *Some T* = *compute_app* (*appST s T1*) *lm* (*appST s T2*).
Lemma *preservation_under_subst_alg* : ∀ {Γ *x s e T1′ t T2 T1 c c′*},
    (*add x* (*SchemeAlt* (*AltSimpTy t*)) Γ) ⊢*a e* : *T1* ! *c*
  → *T1* ~ *T1′*
  → Γ ⊢*a s* : *T2* ! *c′*
  → *compat T1′ c′*
  → (*AltSimpTy t*) ~ *T2*
  → ∃ *c′′*, ∃ *T′′*,
      c′′ ⊆ (*set_union eq_label c c′*) ∧ *T′′* ~ *T1′* ∧ Γ ⊢*a* (*subst_r s x e*) : *T′′* ! *c′′*.
Lemma *preservation_under_lsubst_alg* : ∀ {Γ *r T c v l t*},
    Γ ⊢*a r* : *T* ! *c*
  → *empty* ⊢*a v* : (*AltSimpTy t*) ! *nil*
  → *lookup l t Δ*
  → ∃ *c′*, *c′* ⊆ *c* ∧ Γ ⊢*a* (*subst_l_r v l r*) : *T* ! *c′*.

The remaining definitions, along with some additional proofs, can be found at http://www.doc.ic.ac.uk/~pm1108/kappaproofs.

B.7 Auxiliary Systems

B.7.1 Runtime Expression Type System

$\boxed{\Gamma \vdash_r r : T ! c}$

$$\frac{x : t \in \Gamma}{\Gamma \vdash_r x : t ! \varnothing} \quad \textsc{RAx}$$

$$\frac{\Gamma, x : t \vdash_r e : T ! c}{\Gamma \vdash_r \lambda x.e : t \underline{\rightarrow} T ! c} \quad \textsc{RLam}$$

$$\frac{l : t \in \Delta}{\Gamma \vdash_r ?l : t ! \varnothing} \quad \textsc{RLabel}$$

$$\frac{\begin{array}{l}\Gamma \vdash_r e : T ! c \\ l : t \in \Delta\end{array}}{\Gamma \vdash_r l[e] : T ! c \cup \{l\}} \quad \textsc{RCTrans}$$

$$\frac{\begin{array}{l}\Gamma \vdash_r e_1 : T_1 ! c_1 \\ \Gamma \vdash_r e_2 : T_2 ! c_2 \\ T' = T_1 \cdot T_2 \\ compat(T_1, c_2) \\ compat(T_2, c_1)\end{array}}{\Gamma \vdash_r e_1\, e_2 : T' ! c_1 \cup c_2} \quad \textsc{RApp}$$

$$\frac{\begin{array}{l}l : t \in \Delta \\ \overline{\vdash_v v_i : t}^{\,i} \\ \overline{\Gamma \vdash_r e_i : T_i ! c_i}^{\,i} \\ c' = \cup \overline{c_i}^{\,i} \\ exhaustive(\overline{v}, t)\end{array}}{\Gamma \vdash_r \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow e_i}^{\,i}\} : \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i}^{\,i}\} ! c'} \quad \textsc{RCases}$$

$$\frac{\begin{array}{l}l : t \in \Delta \\ \overline{\vdash_v v_i : t}^{\,i} \\ \overline{\Gamma \vdash_r e_i : T_i ! c_i}^{\,i} \\ \vdash_v v : t \\ c' = \cup \overline{c_i}^{\,i} \\ exhaustive(\overline{v}, t)\end{array}}{\Gamma \vdash_r \mathbf{ccase}\, l\, v\, \mathbf{of}\, \{\overline{v_i \Rightarrow e_i}^{\,i}\} : \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i}^{\,i}\} ! c'} \quad \textsc{RLCases}$$

$$\frac{\begin{array}{l}\Gamma \vdash_r e : T ! c \\ T \sim T'\end{array}}{\Gamma \vdash_r e : T' ! c} \quad \textsc{RSubst}$$

B.7.2 Syntax-directed Type System

$\boxed{\Gamma \vdash_a r : T ! c'}$

$$\frac{x : t \in \Gamma}{\Gamma \vdash_a x : t ! \varnothing} \quad \textsc{AAx}$$

$$\frac{\Gamma, x : t \vdash_a r : T ! c}{\Gamma \vdash_a \lambda x.r : t \underline{\rightarrow} T ! c} \quad \textsc{ALam}$$

$$\frac{l : t \in \Delta}{\Gamma \vdash_a ?l : t ! \varnothing} \quad \textsc{ALabel}$$

$$\frac{\begin{array}{l}\Gamma \vdash_a r : T ! c \\ l : t \in \Delta\end{array}}{\Gamma \vdash_a l[r] : T ! c \cup \{l\}} \quad \textsc{ACTrans}$$

$$\frac{\begin{array}{l}\Gamma \vdash_a r_1 : T_1 ! c_1 \\ \Gamma \vdash_a r_2 : T_2 ! c_2 \\ T' = T_1' \cdot T_2' \\ T_1' \sim T_1 \\ T_2' \sim T_2 \\ compat(T_1', c_2) \\ compat(T_2', c_1)\end{array}}{\Gamma \vdash_a r_1\, r_2 : T' ! c_1 \cup c_2} \quad \textsc{AApp}$$

$$\frac{\begin{array}{l}l : t \in \Delta \\ \overline{\vdash_v v_i : t}^{\,i} \\ \overline{\Gamma \vdash_a r_i : T_i ! c_i}^{\,i} \\ c' = \cup \overline{c_i}^{\,i} \\ exhaustive(\overline{v}, t)\end{array}}{\Gamma \vdash_a \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow r_i}^{\,i}\} : \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i}^{\,i}\} ! c'} \quad \textsc{ACases}$$

$$\frac{\begin{array}{l}l : t \in \Delta \\ \overline{\vdash_v v_i : t}^{\,i} \\ \overline{\Gamma \vdash_a r_i : T_i ! c_i}^{\,i} \\ \vdash_v v : t \\ c' = \cup \overline{c_i}^{\,i} \\ exhaustive(\overline{v}, t)\end{array}}{\Gamma \vdash_a \mathbf{ccase}\, l\, v\, \mathbf{of}\, \{\overline{v_i \Rightarrow r_i}^{\,i}\} : \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i}^{\,i}\} ! c'} \quad \textsc{ALCases}$$

B.7.3 Syntax-directed Polymorphic Type System

$\boxed{\Gamma \vdash_{Pa} r : T\,!\,c'}$

$$\frac{\begin{array}{l} x : \sigma \in \Gamma \\ \sigma \preccurlyeq t \end{array}}{\Gamma \vdash_{Pa} x : t\,!\,\varnothing}\ \textsc{PAAx}$$

$$\frac{\Gamma, x : t \vdash_{Pa} r : T\,!\,c}{\Gamma \vdash_{Pa} \lambda x.r : t \underline{\rightarrow} T\,!\,c}\ \textsc{PALam}$$

$$\frac{l : t \in \Delta}{\Gamma \vdash_{Pa} ?l : t\,!\,\varnothing}\ \textsc{PALabel}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{Pa} r : T\,!\,c \\ l : t \in \Delta \end{array}}{\Gamma \vdash_{Pa} l[r] : T\,!\,c \cup \{l\}}\ \textsc{PACTrans}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{Pa} r_1 : T_1\,!\,c_1 \\ \Gamma \vdash_{Pa} r_2 : T_2\,!\,c_2 \\ T' = T_1' \cdot T_2' \\ T_1' \sim T_1 \\ T_2' \sim T_2 \\ compat(T_1', c_2) \\ compat(T_2', c_1) \end{array}}{\Gamma \vdash_{Pa} r_1\, r_2 : T'\,!\,c_1 \cup c_2}\ \textsc{PAApp}$$

$$\frac{\begin{array}{l} l : t \in \Delta \\ \overline{\vdash_v v_i : t}^{\,i} \\ \overline{\Gamma \vdash_{Pa} r_i : T_i\,!\,c_i}^{\,i} \\ c' = \cup \overline{c_i}^{\,i} \\ exhaustive(\overline{v}, t) \end{array}}{\Gamma \vdash_{Pa} \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow r_i}^{\,i}\} : \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i}^{\,i}\}\,!\,c'}\ \textsc{PACases}$$

$$\frac{\begin{array}{l} l : t \in \Delta \\ \overline{\vdash_v v_i : t}^{\,i} \\ \overline{\Gamma \vdash_{Pa} r_i : T_i\,!\,c_i}^{\,i} \\ \vdash_v r : t \\ c' = \cup \overline{c_i}^{\,i} \\ exhaustive(\overline{v}, t) \end{array}}{\Gamma \vdash_{Pa} \mathbf{ccase}\, l\, r\, \mathbf{of}\, \{\overline{v_i \Rightarrow r_i}^{\,i}\} : \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{v_i \Rightarrow T_i}^{\,i}\}\,!\,c'}\ \textsc{PALCases}$$

B.7.4 Expanded Inference for the Polymorphic System (with explicit fresh variable tape, F)

$\boxed{inf(F, \Gamma, e) = \langle F', S, T, c\rangle}$

$$\frac{\begin{array}{l} x : \sigma \in \Gamma \\ \langle t, F'\rangle = inst(F, \sigma)\end{array}}{inf(F, \Gamma, x) = \langle F', \mathbf{id}, t, \varnothing\rangle}\ \textsc{IAx}$$

$$\frac{\begin{array}{l} inf(F, \Gamma, x : X, e) = \langle F', S, T, c\rangle \\ \mathbf{fresh}\ X\end{array}}{inf(X :> F, \Gamma, \lambda x.e) = \langle F', S, S(X) \underline{\rightarrow} T, c\rangle}\ \textsc{ILam}$$

$$\frac{\begin{array}{l} inf(F, \Gamma, e_1) = \langle F', S_1, T_1, c_1\rangle \\ inf(F', S_1(\Gamma), e_2) = \langle F'', S_2, T_2, c_2\rangle \\ \langle T', S\rangle = S_2(T_1) \cdot_{\mathscr{U}} T_2 \\ compat(S(S_2(T_1)), c_2) \\ compat(S(T_2), c_1) \\ S' = S \circ S_2 \circ S_1\end{array}}{inf(F, \Gamma, e_1\, e_2) = \langle F''', S', S(T'), c_1 \cup c_2\rangle}\ \textsc{IApp}$$

$$\frac{l : t \in \Delta}{inf(F, \Gamma, ?l) = \langle F, \mathbf{id}, t, \varnothing\rangle}\ \textsc{ILabel}$$

$$\frac{\begin{array}{l} inf(F, \Gamma, e) = \langle F', S, T, c\rangle \\ l : t \in \Delta\end{array}}{inf(F, \Gamma, l[e]) = \langle F', S, T, c \cup \{l\}\rangle}\ \textsc{ICTrans}$$

$$\frac{\begin{array}{l} l : t \in \Delta \\ inf i(F_i, \Gamma, ev_i) = \langle F_i', s_i, t\rangle \wedge inf i(F_i'', H, e_i) = \langle F_{i+1}, s_i', T_i\rangle \\ c' = \cup\, \overline{c_i}^{\,i} \\ S'' = (\sqcup\, \overline{S_i}^{\,i}) \sqcup (\sqcup\, \overline{S_i'}^{\,i}) \\ exhaustive(\overline{ev_i\, e_i}^{\,i}, t) \\ v_i = v_i'\end{array}}{inf(F, \Gamma, \mathbf{ccase}\, l\, \mathbf{of}\, \{\overline{ev_i \Rightarrow e_i}^{\,i}\}) = \langle F', S'', \mathbf{Case}\, l\, \mathbf{of}\, \{\overline{ev_i \Rightarrow T_i}^{\,i}\}, c'\rangle}\ \textsc{ICases}$$

$$\frac{inf(F, \Gamma, e) = \langle F', S, T, c\rangle}{inf(F, \Gamma, e) = \langle \mathbf{pop}\, nn\, F', S, T, c\rangle}\ \textsc{IPop}$$

B.8 Proofs

B.8.1 Soundness and completeness of $\vdash_a$

```
Theorem alg_soundness : ∀ {Γ e T c},
  Γ ⊢a e : T ! c → Γ ⊢ e : (SchemeAlt T) ! c.
Proof.
intros Γ e T c H.
```

We proceed by induction on the typing derivation. All cases where the rule from $\vdash_a$ has an equivalent rule in $\vdash_r$ are trivial. The only remaining case is AApp.

```
induction_KTypA H; eauto with kappa_ty_ex.
 Case "AAx".
 apply Inst with (ts' := ts''); simpl; auto.
 apply Ax; auto.
 Case "AApp".
```

We can use the Subst rule on the $\sim$ premises, to make the types match and can then simply apply App.

```
 apply Subst with (T' := T1') in IHKTyp1; auto with relations.
 apply Subst with (T' := T2') in IHKTyp2; auto with relations.
 apply App with (T1 := T1') (T2 := T2'); auto.
Qed.
```

```
Theorem alg_completeness : ∀ {Γ e c ts},
  wf_env Γ →
    Γ ⊢ e : ts ! c → ∃ T, ∃ T',
       Γ ⊢a e : T ! c
    ∧ (gen Γ T') ≤ ts
    ∧ T' ~ T.
Proof.
intros Γ e c ts Hwf H.
```

We prove this Theorem by induction on the derivation of $\Gamma \vdash_r r : T ! c$.

```
induction_KTyp H.
```

Case Ax *Case* "Ax".

$r = x$, $T = t$ and $x : t \in \Gamma$. We can choose T' to be T. We can then apply AAx to get the left side of the conjunction. The right side holds by reflexivity of $\sim$.

```
pose (F := all_vars).
do 2 ∃ (snd (inst F ts')).
splits.
 unfold wf_env in Hwf; destruct (Hwf F x' ts' Hlookup) as (t, Ht); rewrite Ht.
 apply PAAx with (ts5 := ts'); auto.

  case_eq (inst F ts').
  intros F' T' Heq. simpl in *. apply sym_eq in Heq. apply inst_prec in Heq; auto.

 rewrite ← Ht; apply inst_prec2.
 apply gen_inst_prec.
 auto with relations.
```

Case LAM *Case* "Lam".

$r = \lambda x.r'$ and $T = t_1 \underline{\rightarrow} T_2$. By the induction hypothesis (IH) we know that there is a T_2' such that $\Gamma, x : t_1 \vdash_a r' : T_2' ! \, c$ and $T_2 \sim T_2'$. We can thus pick T' to be $t_1 \underline{\rightarrow} T_2'$.

```
destruct IHKTyp as [T'' [T''' [Htyp [Hprec Hsym]]]].
apply wf_env_add_AltSimpTy; auto.
∃ (arrT t' T''); ∃ (arrT t' T''').
splits.
```

We can then apply ALAM to get the left hand side of the conjunction.

```
apply PALam; auto.
apply gen_SchemeAlt_prec_eq in Hprec; rewrite Hprec; apply gen_SchemeAlt_prec.
```

$t_1 \underline{\rightarrow} T_2 \sim t_1 \underline{\rightarrow} T_2'$ follows by Lemma 6.1.

```
rewrite Hsym; reflexivity.
```

Case LBL *Case* "Lbl".

$r = ?l$ and $T = t$ with $l : t \in \Delta$. This case is trivial, as Δ remains the same, so we can just apply ALBL and reflexivity of $\sim$ to get the conclusion.

```
do 2 ∃ (AltSimpTy t').
splits; eauto using gen_SchemeAlt_prec with kappa_ty_alg relations.
```

Case CTRANS *Case* "CTrans".

$r = l[r']$. This case is also trivial, as we know from the IH that there is a T'' such that $\Gamma \vdash_a r' : T'' ! \, c$ with $T'' \sim T$, so we can just choose T' to be T'' and apply CTRANS. The $\sim$ part of the conclusion matches the one in the IH.

```
destruct (IHKTyp Hwf) as [T'' [T''' [Htyp [Hprec Hsym]]]].
```

∃ T''; ∃ T'''.
split.
 apply *PACTrans* with (*t5* := t').
  apply *all_T_are_T*.
  auto.
  auto.
 auto.

Case App *Case* "App".

$r = r_1 r_2$ and $T = T_1 \cdot T_2$ with $\Gamma \vdash_r r_1 : T_1 ! c_1$, $\Gamma \vdash_r r_2 : T_2 ! c_2$, *compat*$(T_1, c_2)$, and *compat*$(T_2, c_1)$. From the IH we know:

- there is a T_1' such that $\Gamma \vdash_a r_1 : T_1' ! c_1$ and $T_1' \sim T_1$, and
- there is a T_2' such that $\Gamma \vdash_a r_2 : T_2' ! c_2$ and $T_2' \sim T_2$.

We can choose $T' = T$, and we have everything we need to apply AApp. The $\sim$ part of the conclusion follows by reflexivity.

destruct (*IHKTyp1 Hwf*) as (*T1'*, (*T1''*, (*HTyp1*, (*HsubtypT1*, *HsimT1*)))).
destruct (*IHKTyp2 Hwf*) as (*T2'*, (*T2''*, (*HTyp2*, (*HsubtypT2*, *HsimT2*)))).
assert (∀ *a b*, *a* ⊑ *b* → *a* ∼ *b*) as *Hleq_red_impl_simT*.
  intros; apply *t_step*; unfold *simT1*; auto.
  ∃ T'; ∃ T'.
  splits.
   apply *PAApp* with (*T1* := *T1'*) (*T2* := *T2'*) (*T1'* := *T1*) (*T2'* := *T2*);
    eauto using *all_T_are_T* with *relations*.
   apply *gen_SchemeAlt_prec_eq* in *HsubtypT1*; rewrite ← *HsubtypT1*; auto.
   apply *gen_SchemeAlt_prec_eq* in *HsubtypT2*; rewrite ← *HsubtypT2*; auto.
   apply *gen_SchemeAlt_prec*.
   reflexivity.

Cases CCases and LCases: *Case* "Cases".

$r =$ **ccase** l **of** $\{\overline{v \Rightarrow r}\}$ and $T =$ **Case** l **of** $\{\overline{v \Rightarrow T}\}$. From the IH we know that:

- there are $\overline{T''}$ such that

$$\overline{\Gamma \vdash_a r_i : T_i'' ! c_i}^{\,i} \wedge \overline{T'' \sim T}$$

destruct (*In_exists_list HKTyp_rrTcs2*) as (*f*, *Hf*).

```
pose (Tr_ :=
        (AltAlt l'
         (map
            (fun pat_ : r × r × T × c ⇒
             let (p, _) := pat_ in
             let (p0, T_) := p in let (v_, _) := p0 in v_  T_) rrTcs))).
```

We can thus choose T' to be **Case** l **of** $\{\overline{v \Rightarrow T''}\}$. For the right side we can apply AACases or ALCases, respectively.

```
eexists; ∃ Tr_.
splits.
 apply (independence_rrTcs_ACases f).
 apply PACases with (t5 := t'); try solve [destruct rrTcs in *; ssimpl; auto]; intros.
 clear Hv_rrTcs HT_rrTcs Hlookup HKTyp_rrTcs1_v HKTyp_rrTcs1
  HKTyp_rrTcs2 Hunion Hexhaust Hf.
 induction rrTcs; auto.
  simpl; apply Forall_nil.
  simpl; apply Forall_cons; auto.
   destruct a in *; destruct p in *; destruct p in *; apply all_T_are_T.
 destruct (replace_T_only_T H) as [T_' HIn].
 eapply HKTyp_rrTcs1_v; eauto.
 intros; destruct (replace_T_only_T H) as [T_' HIn]; destruct (Hf v_ r_ T_' T_ Tr_ c_) as
(_, (Htyp, (Hsubtyp, Hsim))); auto.
 unfold replace_T; ssimpl; rewrite Hunion; reflexivity.
 unfold replace_T; ssimpl; auto.
 clear Hv_rrTcs HT_rrTcs Hlookup HKTyp_rrTcs1_v HKTyp_rrTcs1 HKTyp_rrTcs2
  Hunion Hexhaust.
 apply gen_SchemeAlt_prec.
```

$T \sim T'$ follows by Lemma 6.2.

```
 apply simT_AltAlt.
 intros.
 destruct (Hf v_ r_ T_ (f v_ r_ T_ c_) Tr_ c_) as (Hsub, (Htyp, (Hsubtyp, Hsim))); auto.
 apply gen_SchemeAlt_prec_eq in Hsubtyp; rewrite Hsubtyp in Hsim; eauto with relations.
Case "LCases".
destruct (In_exists_list HKTyp_rrTcs2) as (f, Hf).
pose (Tr_ :=
```

```
        (AltAlt l'
         (map
            (fun pat_ : r × r × T × c ⇒
              let (p, _) := pat_ in
              let (p0, T_) := p in let (v_, _) := p0 in v_  T_) rrTcs))).
  eexists; ∃ Tr_.
  splits.
   apply (independence_rrTcs_ALCases f).
   apply PALCases with (t5 := t'); try solve [destruct rrTcs in *; ssimpl; auto]; intros.
   clear Hv_rrTcs HT_rrTcs Hlookup HKTyp_rrTcs1_v HKTyp_rrTcs1
    HKTyp_rrTcs2 Hunion Hexhaust Hf.
   induction rrTcs; auto.
    simpl; apply Forall_nil.
    simpl; apply Forall_cons; auto.
     destruct a in *; destruct p in *; destruct p in *; apply all_T_are_T.
   destruct (replace_T_only_T H) as [T_' HIn].
   eapply HKTyp_rrTcs1_v; eauto.
   intros; destruct (replace_T_only_T H) as [T_' HIn]; destruct (Hf v_ r_ T_' T_ Tr_ c_) as
(_, (Htyp, (Hsubtyp, Hsim))); auto.
   unfold replace_T; ssimpl; rewrite Hunion; reflexivity.
   unfold replace_T; ssimpl; auto.
   clear Hv_rrTcs HT_rrTcs Hlookup HKTyp_rrTcs1_v HKTyp_rrTcs1 HKTyp_rrTcs2
    Hunion Hexhaust.
   apply gen_SchemeAlt_prec.
   apply simT_AltAlt.
   intros.
   destruct (Hf v_ r_ T_ (f v_ r_ T_ c_) Tr_ c_) as (Hsub, (Htyp, (Hsubtyp, Hsim))); auto.
   apply gen_SchemeAlt_prec_eq in Hsubtyp; rewrite Hsubtyp in Hsim; eauto with relations.
```

Case Subst *Case* "Subst".

From the premises of this case we know there is T'' such that $T'' \sim T$. From the IH we also know that there is a T''' such that $\Gamma \vdash_a r : T'''!c$ and $T''' \sim T''$. We can thus pick T' to be T'''. The typing part of the conclusion follows directly from the IH, and the $\sim$ part follows by transitivity of $\sim$, as $T \sim T'' \wedge T'' \sim T''' \Rightarrow T \sim T'''$ as required.

```
  destruct (IHKTyp Hwf) as [Tf [Tfr [HTyp [Hsubtyp HsimT']]]].
  ∃ Tf; ∃ T'; splits; auto.
```

```
    apply gen_SchemeAlt_prec.
    apply gen_SchemeAlt_prec_eq in Hsubtyp; rewrite Hsubtyp in *; apply symmetry in HsimT;
eapply transitivity; eauto .

  Case "Inst".
  destruct (IHKTyp Hwf) as [Tf [Tfr [HTyp [Hsubtyp' HsimT]]]].
  eexists; eexists; splits; eauto.
  eapply transitivity; eauto.

  Case "Gen".
  destruct (IHKTyp Hwf) as [Tf [Tfr [HTyp [Hsubtyp' HsimT]]]].
  eexists; eexists; splits; eauto.
  apply Forall_prec_gen; auto.
Qed.
```

B.8.2 Type soundness

```
Theorem progress : ∀ Γ r ts c Θ,
      wf Θ
  → nil = fl r
  → Γ = empty
  → Γ ⊢ r : ts ! c
  → Is_true (is_v_of_r r) ∨ ∃ r', ∃ Θ', «r, Θ –>κ r', Θ'».
Proof.
intros Γ e ts c Θ H0 H1 H2_Γ H2.
```

We know from $\vdash r : T \downarrow \varnothing ! c$ that $\vdash r : T ! c$ and $fl(r) = \varnothing$. We proceed by induction over the derivation of $\vdash r : T ! c$. For Lam, Label and Cases, $r \in Val$.

```
induction_KTyp H2; auto; try solve [ left; simpl; auto ].
```

Case Ax *Case* "Ax".

We know that $\Gamma = \varnothing$, so this case is impossible.

```
  compute in Hlookup; rewrite H2_Γ in Hlookup; simpl in Hlookup; destruct Hlookup.
```

Case CTrans *Case* "CTrans".

```
  case_eq (Θ l'); intros v vs HΘ.
```

We can apply Kappa.

```
  right; do 2 eexists; eapply Kappa.
    2: unfold pop; rewrite HΘ; reflexivity.
    unfold wf in *.
```

```
case (H0 l' t'); auto.
intros Hwfv Hwfvs; destruct Hwfv as [Hwv_v Hwv_ty].
rewrite HΘ in Hwv_v; simpl in Hwv_v; auto.
```

Case App *Case* "App".

```
apply eq_sym in H1.
simpl in H1.
apply union_eq_nil in H1.
destruct H1 as [H1_1 H1_2].
```

Follows from the IH combined with AppArg/AppFun if either r_1 or r_2 are not values. Otherwise:

```
destruct IHKTyp1, IHKTyp2; auto.
```

SCase "r1 Val r2 Val".

```
induction r1; try (solve [ destruct H1 ]).
```

- If $r_1 = \lambda x.r_1'$, we can apply Beta.

  SSCase "r1 = λx, r1".

  ```
  right; do 2 eexists; eapply Beta.
  ```

- If $r_1 =$ **ccase** l **of** $\{\cdots\}$ or $?l$, $fl(r_1\, r_2) \neq \emptyset$, so this case is impossible.

  SSCase "r1 = ?l".

  ```
  simpl in H1_1; inversion H1_1.
  ```

  SSCase "r1 = ccase l of {}".

  ```
  simpl in H1_1; destruct (union_cons_eq_nil _ _ H1_1).
  ```

SCase " r2', r2 –>κ r2'".

```
destruct H4 as [r2' [Θ' Hr2]].
right; do 2 eexists; eapply AppArg; eapply Hr2.
```

SCase " r1', r1 –>κ r1'".

```
destruct H1 as (r1', (Θ', Hr1)).
right; do 2 eexists; eapply AppFun; auto; eapply Hr1.
```

SCase " r1' and r2'".

```
destruct H4 as [r2' [Θ' Hr2]].
right; do 2 eexists; eapply AppArg; eapply Hr2.
```

Case "Cases".

```
left; apply ott_forall_coq_Forall; ssimpl; auto.
```

Case LCases *Case* "LCases".
From the *exhaustive*(⋯) premise we know that typeable **ccase** terms are exhaustive, so $\exists i.v = vi$, and we can apply CCase.

```
pose (Hirex := exhaustive_def Hexhaust).
destruct (Hirex r' HKTyp) as [i [r'' Hex]].
right; do 2 eexists; eapply CCase; ssimpl; eauto.
  apply map_nth_error with (f := @fst r r) (B := r) in Hex.
  ssimpl in Hex.
  eapply (nth_error_Forall_inv (fun x ⇒ Is_true (is_v_of_r x))).
  eapply Hex.
  ssimpl; auto.
```

Case Subst Follows directly from the IH.

```
Qed.

Theorem preservation_alg: ∀ {Γ Θ e Θ' e' T c},
      wf_env Γ
  → wf Θ
  → Γ ⊢a e : T ! c
  → «e, Θ –>κ e', Θ'»
  → ∃ c', ∃ T', ∃ T'', c' ⊆ c ∧ T' ≤ T ∧ T'' ~ T' ∧ Γ ⊢a e' : T'' ! c'.
Proof.
intros Γ Θ e Θ' e' T c Henv H0 H1 H2.
```

By induction on the derivation of $e, \Theta \longrightarrow e', \Theta'$ `generalize dependent` *H1*.

```
dependent induction H2 generalizing T c; try (rename H1 into H2 || rename H1 into H1'); intro
H1.
```

Case Beta *Case* "Beta".
Holds by preservation under substitution, as stated in Lemma 6.8. `inversion` *H1*.

```
  inversion H9.
  rewrite ← H24 in *.
  destruct (simT_arrT_simpl H12 H11) as (HT1', HT2).
  symmetry in H12.
  destruct (arrT_simT_determines H12) as (Tx, ->).
  apply compat_arrT in H15.
  apply arrT_simT_pairwise in H12.
  destruct
    (preservation_under_subst_alg H26 H12 H10 H15 (transitivity HT2 H13))
```

as (*c'f*, (*T'f*, (*Hsub*, (*HsimT*, *Htyp*)))).
∃ *c'f*; ∃ *T*; ∃ *T'f*.
splits; auto with *relations*.
symmetry in *H12*.
exact (transitivity (transitivity *HsimT H12*) *HT1'*).

Case Kappa *Case* "Kappa".
Holds by preservation under label substitution, as stated in Lemma 6.9. inversion *H1*.
pose (*@wf_pop_ty_alg _ _ _ _ _ H0 H2 H11*) as *Ht*.
destruct (*preservation_under_lsubst_alg H8 Ht H11*) as [*c'* [*Hsub Htyp*]].
∃ *c'*; ∃ *T*; ∃ *T*; *splits*; auto with *relations*.
rewrite *Hsub*; apply *subset_union_l*.

Case CCase *Case* "CCase".
By inverting $\Gamma \vdash_a r : T ! c$, we can then take $T' = T_i$. $T_i \leq T$ follows by the definition of $\leq$.
inversion *H1*.
case *t102* in *.
apply *nth_error_in* in *H1'*.
destruct (*v_r_in_v_r_T_c H1' H8*) as (*Ti*, (*ci*, *Hin*)).
pose (*Hvrtc* := *v_r_T_c_in_v_T Hin*).
pose (*Hin'* := *leq_case l5 Hvrtc*).
∃ *ci*; ∃ *Ti*; ∃ *Ti*.
splits; auto with *relations*.
eapply *ci_in_union*; eauto.
eauto.

Case AppArg *Case AppArg*.
By inverting $\Gamma \vdash_a r : T ! c$ we get:

$$\frac{\begin{array}{l}\Gamma \vdash_a r_1 : T_1 ! c_1 \\ \Gamma \vdash_a r_2 : T_2 ! c_2 \\ T' = T_1' \cdot T_2' \\ T_1' \sim T_1 \\ T_2' \sim T_2 \\ \mathit{compat}(T_1', c_2) \\ \mathit{compat}(T_2', c_1)\end{array}}{\Gamma \vdash_a r_1\, r_2 : T' ! c_1 \cup c_2}\ \textsc{AApp}$$

inversion *H1*.

From the IH we get that there is a c_3 and T_3 such that:

- $c_3 \subseteq c_2$
- $T_3 \leq T_2$
- $\Gamma \vdash_a r_2 : T_3 ! c_3$

`destruct` (*IHKOp H0 T2 c2 H11*) `as` [*c3* [*T3* [*T3''* [*Hsub* [*Hleq* [*HsimT Hty*]]]]]].
Thus, we can take c' to be $(c_1 \cup c_3)$. We can then apply AApp if we can prove:

- $\Gamma \vdash r_1 : T_1 ! c_1$: Follows from the inversion.
- $\Gamma \vdash r_2 : T_2 ! c_2$: Follows from the IH.
- $\exists T_1'' T_2'' T_f . T_1'' \sim T_1 \wedge T_2'' \sim T_2 \wedge T_f = T_1'' \cdot T_2''$:

We know from the IH that $T_3 \leq T_2$. This means that the set of labels used to determine the type of T_3 has to be a subset of the set of labels used to determine the type of T_2. Hence, we can add just enough redundant labels to T_3 to match the labels that were removed. This means that there is a T_3', which is $\sim$-equivalent to T_3, such that $T_1 \cdot T_3'$ is defined, with a type that is smaller than the original resulting type. So, we can take T_1'' to be T_1 and T_2'' to be T_3'.

- $compat(T_1, c_3)$: Follows from $c_3 \subseteq c_2$ and the definition of compat.
- $compat(T_3', c_1)$: This holds as T_3' uses the same set of labels to determine its type as T_2, and we know $compat(T_2, c_1)$ from the IH.

∃ (*set_union eq_label c1 c3*).
`destruct` (*compat_determines_labels H12 H19 H14 Hleq*) `as` [*T3'* [*Tf'* [*HsimT3'* [*HappT3'* [*HcompatT3' HsimTf'*]]]]].
`do` 2 (∃ *Tf'*).
splits; `auto with` *relations*.
`eapply transitivity`; `eauto`.
`rewrite` *Hsub*; `reflexivity`.
`reflexivity`.
`eapply` *PAApp*; `auto with` *relations*; `eauto with` *relations*.
`eapply transitivity`; `eauto with` *relations*.
`rewrite` *Hsub*; `auto`.

Case AppFun *Case AppFun*.
Holds by a similar argument as case AppArg `inversion` *H1*.

```
  destruct (IHKOp H0 T1 c1 H11) as [c3 [T3 [T3'' [Hsub [Hleq [HsimT Hty]]]]]].
  ∃ (set_union eq_label c3 c2).
  destruct (compat_determines_labels2 H13 H17 H14 Hleq) as [T3' [Tf' [HsimT3' [HappT3' [HcompatT3' HsimTf']]]]].
  do 2 (∃ Tf').
  splits; auto with relations.
   eapply transitivity; eauto.
   rewrite Hsub; reflexivity.
   reflexivity.
   eapply PAApp; auto with relations; eauto with relations.
   eapply transitivity; eauto with relations.
   rewrite Hsub; auto.
Qed.
Theorem preservation: ∀ {Γ Θ e Θ' e' ts c F},
      wf Θ
  → Γ ⊢ e : ts ! c
  → «e, Θ –>κ e', Θ'»
  → wf_env Γ
  → ∃ c', ∃ ts',
          c' ⊆ c
      ∧ ((@snd Kappa.F Kappa.T) (inst F ts')) ≤ ((@snd Kappa.F Kappa.T) (inst F ts))
      ∧ Γ ⊢ e' : ts' ! c'.
Proof.
```

Straightforward by combining soundness and completeness of $\vdash_a$ with type preservation for $\vdash_a$ and the auxiliary Lemmas for the interaction between $\leq$ and $\sim$: We can apply Theorem 6.2 (Completeness of $\vdash_a$) to the typing judgement to get:

$$\exists T''.\Gamma \vdash_a e : T''! c \wedge T \sim T''$$

```
  intros.
  eapply (@inst_KTyp Γ e ts c F) in H0.
  destruct (@alg_completeness Γ e c (SchemeAlt (snd (inst F ts))) H2 H0) as [T' [Ti [HTyp [Hsubtyp HsimT]]]].
```

We can then apply Theorem 6.4 (Type preservation for $\vdash_a$) to get:

$$\exists c'T'.c' \subseteq c \wedge T' \leq T'' \wedge \Gamma \vdash_a e' : T'! c' \wedge T \sim T''$$

```
  destruct (preservation_alg H2 H HTyp H1) as [c'a [T'a [T''a [Hcsuba [Hleqa [HsimTa Htypa]]]]]].
```

Applying Theorem 6.1 (Soundness of $\vdash_a$) to the typing judgement then yields:

$$\exists c' T'. c' \subseteq c \wedge T' \leq T'' \wedge \Gamma \vdash e' : T' ! c' \wedge T \sim T''$$

`apply` *alg_soundness* `in` *Htypa*.
`apply symmetry in` *HsimT*.

By Lemma 6.5, we know that: $\exists T_f. T_f \sim T'' \wedge T' \leq T_f$.

`destruct` (*almost_proper_simT_leq HsimT Hleqa*) `as` [*Tf* [*Hleqf HsimTf*]].
∃ *c'a*; ∃ (*SchemeAlt Tf*); `auto`.
`simpl`; `case` *F* `in` *; `auto`.

splits; `auto with` *relations*.
`apply` *gen_SchemeAlt_prec_eq* `in` *Hsubtyp*; `rewrite` ← *Hsubtyp* `in` *; `simpl`; `auto`.

By transitivity of $\sim$, $T \sim T_f$, so we can finally apply Subst to get:

$$\exists c' T'. c' \subseteq c \wedge T' \leq T \wedge \Gamma \vdash e' : T' ! c'$$

`eapply` *Subst*; `eauto with` *relations*.
`eapply transitivity`; `eauto with` *relations*.
`Qed`.

B.8.3 Soundness of inference

`Theorem` *inf_a_soundness* : ∀ *F Γ e F' s T c*,
inf(*F*, *Γ*, *e*) = « *F'*, *s*, *T*, *c* » → (*appSΓ s Γ*) ⊢*a e* : *T* ! *c*.
`Proof`.
`intros`.
By induction over the derivation of $inf(\Gamma, e) = \langle S, T, c \rangle$ `induction` *H*; `eauto with` *kappa_ty_alg*.

Case IAx *Case* "IAx".
We can apply AAx. `eapply` *PAAx*; `eauto`; [`apply` *appSΓ_nil_id* | `eapply` *inst_prec*]; `eauto`.

Case ILam *Case* "ILam".
$e = \lambda x.e'$ and $T = S(X) \underline{\rightarrow} T'$. By the IH and the definition of substitution we have:

$$S\Gamma, x : S(X) \vdash_a e' : T ! c$$

We can then apply ALam to get:

$$S\Gamma \vdash \lambda x.e' : S(X) \underline{\rightarrow} T' ! c$$

`apply` *PALam*; `auto`.
`rewrite` *appS_dist_add* `in` *IHIInf*; `auto`.

Case IApp *Case* "IApp".
By the IH we have:

- $S_1\Gamma \vdash_a r_1 : T_1 ! c_1$
- $S_2S_1\Gamma \vdash_a r_2 : T_2 ! c_2$

We can apply Lemma 6.12 repeatedly to the previous to get:

- $SS_2S_1\Gamma \vdash_a r_1 : SS_2T_1 ! c_1$
- $SS_2S_1\Gamma \vdash_a r_2 : ST_2 ! c_2$

`apply` *subst_typ* `with` (*s* := *s2*) `in` *IHIInf1*.
`apply` *subst_typ* `with` (*s* := *s_5*) `in` *IHIInf1*.
`apply` *subst_typ* `with` (*s* := *s_5*) `in` *IHIInf2*.
We can then apply AApp to the previous if we can prove:

$$\exists T_1' T_2' T'.T' = T_1' \cdot T_2' \wedge T_1' \sim SS_2T_1 \wedge T_2' \sim ST_2$$

We can choose $T_1' = S(canonical(S_2T_1))$, and $T_2' = S(inflate(canonical(S_2T_1), canonical(T_2)))$. The $\sim$ parts of the conclusion then follow by $\sim$-equivalence of types after *canonical* and *inflate* (easy to see from the definition of *canonical* and *inflate*). *compat* is preserved over *canonical* and *inflate*, as it is calculated on the *canonical* type, by definition. The fact that $S(canonical(S_2T_1)) \cdot S(inflate(canonical(S_2T_1), canonical(T_2)))$ is defined and equal to T' then follows by Corollary 6.1 (Soundness of $\cdot_{\mathscr{U}}$).

`eapply` *PAApp* `with` (*T1* := *appST s_5* (*appST s2 T1*)) (*T2* := *appST s_5 T2*)
(*T1'* := *appST s_5* (*canonical_dec* (*appST s2 T1*))) (*T2'* := *appST s_5* (*inflate* (*canonical_dec* (*appST s2 T1*)) (*canonical_dec T2*)))
; `eauto with` *relations*; `try` (`rewrite` *H8*); `try` (`do` 2 (`rewrite` ← *appSI_compS*)); `auto`.
`eapply` *compute_appU_soundness*; `eauto`.
`apply` *appST_simT*.
`rewrite` *canonical_simT*.
`reflexivity`.
`rewrite` *inflate_appST*.
`rewrite` *inflate_simT*.

```
  apply appST_simT.
  rewrite canonical_simT.
  reflexivity.

  apply ← (@appST_compat s_5) in H6; apply → (@appST_compat s_5); apply compat_canonical;
auto.

  apply → (@appST_compat s_5); apply compat_inflate; apply ← (@appST_compat s_5) in H7;
apply compat_canonical; auto.
```

Case ICases *Case* "ICases".

```
  iexists3.
  eapply PACases with (v_r_T_c_list :=
    map (fun tmp_ : r × r × k × s × T × s × c ⇒
           let (p, c) := tmp_ in
           let (p0, s') := p in
           let (p1, T) := p0 in
           let (p2, s) := p1 in
           let (p3, k) := p2 in
           let (ev, e) := p3 in (ev, e,
            (appST (fold_left compS
                       (take (List.length v_r_k_s_T_s'_c_list - k - 1)
                           (reverse
                               (map
                                   (fun
                                       pat_0 : Kappa.r × Kappa.r × Kappa.k ×
                                                   Kappa.s × Kappa.T × Kappa.s × Kappa.c ⇒
                                     let (p4, _) := pat_0 in
                                     let (p5, _) := p4 in
                                     let (p6, _) := p5 in
                                     let (p7, s_0) := p6 in
                                     let (p8, _) := p7 in let (_, _) := p8 in s_0)
                                   v_r_k_s_T_s'_c_list))) nil) T), c)) v_r_k_s_T_s'_c_list); ssimpl;
eauto.
      admit.

      intros; apply in_map_iff in H8; destruct H8 as (x, (Hxeq, Hxin));
      destruct x in *; do 5 destruct p in *; injection Hxeq; intros; rewrites;
      exact (H2 v_ r_ k s0 t s c_ Hxin).
```

By the IH we have:

$$\overline{(S_i \circ ... \circ S_1)(\Gamma) \vdash_a r_i : T_i ! c_i}$$

```
intros; apply in_map_iff in H8; destruct H8 as (x, (Hxeq, Hxin));
destruct x in *; do 5 destruct p in *; injection Hxeq; intros; rewrites.
  pose (lfs_IH H3 k v_ t r_ s0 s c_ Hxin) as Hlfs_IH.
```

We can apply Lemma 6.12 with the substitution $(S_n \circ ... \circ S_{i+1})$, yielding:

$$\overline{(S_n \circ ... \circ S_1)(\Gamma) \vdash_a r_i : (S_n \circ ... \circ S_{i+1})(T_i) ! c_i}$$

```
apply (subst_typ
  (fold_left compS
    (take (List.length v_r_k_s_T_s'_c_list - k - 1)
      (reverse
        (map
          (fun
              pat_0 : Kappa.r × Kappa.r × Kappa.k × Kappa.s × T ×
                      Kappa.s × Kappa.c ⇒
            let (p4, _) := pat_0 in
            let (p5, _) := p4 in
            let (p6, _) := p5 in
            let (p7, s_0) := p6 in
            let (p8, _) := p7 in let (_, _) := p8 in s_0)
          v_r_k_s_T_s'_c_list))) nil)) in Hlfs_IH.
rewrite appSΓ_fold_compS in Hlfs_IH; rewrite take_drop_app_fusion in Hlfs_IH; auto.
```

We can then apply ACases to get the desired result as the other premises are also premises of ICases.

```
    splits; ssimpl; try (rewrite length_map_fusion); apply eq_sym; ssimpl; try (rewrite length_map_fusio
reflexivity.
Qed.

Theorem inf_soundness : ∀ F Γ e F' s T c,
  inf(F, Γ, e) = « F', s, T, c » → (appSΓ s Γ) ⊢ e : (SchemeAlt T) ! c.
Proof.
```

Follows trivially by combining the previous restricted soundness Theorem with Theorem 6.1 (Soundness of $\vdash_a$).

```
  intros; apply inf_a_soundness in H; apply alg_soundness in H; auto.
Qed.
```

APPENDIX C

Haskell source code

Context.Utils

```
{-# LANGUAGE MultiParamTypeClasses, TypeFamilies, FlexibleInstances,
             OverlappingInstances, UndecidableInstances, FunctionalDependencies,
             TypeSynonymInstances, TypeOperators, TemplateHaskell #-}
module Context.Utils where

import Data.HList
import Language.Haskell.TH
import Language.Haskell.TH.Quote
import Language.Haskell.Exts.Parser
    (parseTypeWithMode, ParseResult(..), ParseMode(..), defaultParseMode)
import qualified Language.Haskell.Exts.Syntax as H
import Language.Haskell.Exts.Extension
import Language.Haskell.Meta.Syntax.Translate

class HUnion l1 l2 lr | l1 l2 -> lr where
    hUnion :: l1 -> l2 -> lr

instance (HDiff' l2 l1 l1 lr, HAppend l1 lr lr') => HUnion l1 l2 lr' where
    hUnion l1 l2 = hAppend l1 (hDiff' l2 l1 l1)

class HDiff' xs ys oys rs | xs ys oys -> rs where
    hDiff' :: xs -> ys -> oys -> rs

instance HDiff' HNil ys oys HNil where
    hDiff' _ ys oys = hNil

instance (HDiff' xs oys oys rs) => HDiff' (x :*: xs) HNil oys (x :*: rs) where
    hDiff' (HCons x xs) _ oys = (HCons x (hDiff' xs oys oys))

instance (HDiff' xs oys oys rs) => HDiff' (x :*: xs) (x :*: ys) oys rs where
```

```
    hDiff' (HCons x xs) (HCons x' ys) oys = hDiff' xs oys oys

instance (HDiff' (x :*: xs) ys oys rs) => HDiff' (x :*: xs) (y :*: ys) oys rs where
    hDiff' (HCons x xs) (HCons y ys) oys = hDiff' (HCons x xs) ys oys

class HUpdateAtTypeOrAppend e l r | e l -> r where
  hUpdateAtTypeOrAppend :: e -> l -> r

instance HUpdateAtTypeOrAppend e (e :*: l) (e :*: l) where
  hUpdateAtTypeOrAppend e (HCons e' l) = HCons e l

instance HUpdateAtTypeOrAppend e l lr =>
              HUpdateAtTypeOrAppend e (e' :*: l) (e' :*: lr) where
  hUpdateAtTypeOrAppend e (HCons e' l) = HCons e' (hUpdateAtTypeOrAppend e l)

instance HUpdateAtTypeOrAppend e HNil (e :*: HNil) where
  hUpdateAtTypeOrAppend e HNil = e .*. hNil

--  Quasiquoters.

h :: QuasiQuoter
h
  = QuasiQuoter { quoteExp  = undefined
                , quotePat  = undefined
                , quoteType = quoteHType
                , quoteDec  = undefined
                }

quoteHType :: String -> TypeQ
quoteHType s
  = do
      let parseMode = defaultParseMode { extensions = glasgowExts }
          parseResult = parseTypeWithMode parseMode $ "(" ++ s ++ ")"

      case parseResult of
        ParseFailed loc e -> error $ "Parse error: " ++ show (loc, e)
        ParseOk ty        -> toHListTy ty

toHListTy :: H.Type -> TypeQ
toHListTy (H.TyParen ty)
  = [t| $(return . toType $ ty) :*: HNil |]

toHListTy (H.TyTuple _ tys)
  = foldr (appT . appT (conT ''(:*:))) [t| HNil |] . map (return . toType) $ tys

toHListTy t = fail $ "Type malformed: " ++ (show t)
```

Context.Types

```
{-# LANGUAGE OverlappingInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeOperators #-}
module Context.Types where

class BifunctorMixed f where
  mbimap :: (a -> b) -> (d -> c) -> f c a -> f d b

class k :▷ c where
  q :: k -> c

instance (:▷) c c where
  q = id
```

Context.Implicit

```
{-# LANGUAGE FlexibleContexts, TypeOperators,
             QuasiQuotes, GeneralizedNewtypeDeriving #-}
module Context.Implicit where

import Control.Arrow
import Control.Applicative
import Data.List
import Data.HList
import Data.Monoid

import Context.Types
import Context.Utils

newtype ContextF c a = ContextF {runContextF :: c -> a}
  deriving (Functor, Applicative, Monad)

type a :↓ c = ContextF c a

instance BifunctorMixed (->) where
  mbimap f g = \x -> f . x . g

instance BifunctorMixed ContextF where
  mbimap f g (ContextF x) = ContextF (mbimap f g x)

(<^*>) :: (BifunctorMixed f , Applicative (f cr) , HUnion c1 c2 cr
          , cr :▷ c1 , cr :▷ c2)
```

```
       => f c1 (a -> b) -> f c2 a -> f cr b
af <^*> ax = mbimap id q af <*> mbimap id q ax

infixl 4 <^*>

liftAP2 :: (HUnion c1 c2 cr, cr :⊳ c1, cr :⊳ c2)
        => (a -> b -> c) -> a :↓ c1 -> b :↓ c2 -> c :↓ cr
liftAP2 f a b = f <$> a <^*> b
liftAP3 f a b c = f <$> a <^*> b <^*> c

comp :: (HUnion c1 c2 cr, cr :⊳ c1, cr :⊳ c2)
     => (b -> c) :↓ c2 -> (a -> b) :↓ c1
     -> (a -> c) :↓ cr
comp g f = ContextF $
  \cr -> evalC g cr . evalC f cr

evalC :: (c2 :⊳ c1) => a :↓ c1 -> c2 -> a
evalC ca k = ca `runContextF` q k
(*) :: (c2 :⊳ c1) => a :↓ c1 -> c2 -> a
(*) = evalC

mkC0 :: a -> a :↓ HNil
mkC0 = ContextF . const

mkC1 :: (c -> a) -> a :↓ [h| c |]
mkC1 f = ContextF (f . hHead)
ι1 = mkC1

mkC :: (HUnion cs [h| c |] cr , cr :⊳ cs , cr :⊳ [h| c |])
    => (c -> a :↓ cs) -> a :↓ cr
mkC = comb . mkC1
  where
    comb :: (HUnion c1 c2 cr , cr :⊳ c1 , cr :⊳ c2)
         => (a :↓ c1 :↓ c2) -> (a :↓ cr)
    comb cca = ContextF $ \k -> (cca `evalC` k) `evalC` k
ι :: (HUnion cs [h| c |] cr , cr :⊳ cs , cr :⊳ [h| c |])
    => (c -> a :↓ cs) -> a :↓ cr

ι = mkC

-- mkC' :: (HUnion cs [h| c |] cr , cr :⊳ cs , cr :⊳ [h| c |])
--      => (c -> a :↓ cs) -> a :↓ cr
-- mkC' e = ContextF $ \cr -> (e (hHead (q cr))) `runContextF` (q cr)
```

Context.Runtime.PMonad

```
module Context.Runtime.PMonad where

class PMonad m where
  return :: a -> m c c a
  (>>=) :: m c1 c2 a -> (a -> m c2 c3 b) -> m c1 c3 b
```

Context.Runtime.Features

```
module Context.Runtime.Features where

import Context.Knowledge.Features
import Context.Runtime

individual ▸ feat = updateCT (individual :> feat)
infixr 1 ▸
```

Context.Runtime.Realizable

```
{-# LANGUAGE TypeFamilies, TypeOperators #-}
module Context.Runtime.Realizable where

import Prelude hiding ((>>=), (>>))
import Control.Applicative
import Context.Prelude

import Context.Runtime.Streams

import Data.HList hiding ((>>=), (>>))

class Realizable c where
  fetch :: IO c

  realize :: ContextRuntimeT IO HNil c ()
  realizeCF :: a :↓ c -> ContextRuntimeT IO HNil c a

  realize = liftCRT fetch >>= pushCT
  realizeCF x = liftCRT fetch >>= pushCT >> cfToCrT x

instance Realizable HNil where
  fetch = Prelude.return hNil

instance (HList cs, Realizable c, Realizable cs)
          => Realizable (HCons c cs) where
  fetch = hCons <$> fetch <*> fetch

-- class RealizableWith c where
--   type Cfg c :: *
```

```
--   realizeWith :: Cfg c -> a :+ c -> IO a
--   fetchWith :: Cfg c -> IO c
--
--   realizeWith cfg x = fmap (runContextF x) (fetchWith cfg)
```

Context.Runtime.Runtime

```
{-# LANGUAGE TypeOperators, QuasiQuotes, FlexibleContexts #-}
module Context.Runtime.Runtime where

import Prelude hiding (return, (>>=), (>>))
import qualified Control.Monad as M

import Data.HList ( HList, HCons(..), hCons
                  , HNil(..), hNil, (:*:)
                  , (.*.), Fail, TypeNotFound
                  )

import Context.Utils
import Context.Types
import Context.Implicit
import Context.Runtime.PMonad
import Control.Arrow (first)

m1 >> m2 = m1 >>= \_ -> m2

newtype ContextRuntime c1 c2 a =
  CR { runContextRuntime :: c1 -> (a, c2) }

instance PMonad ContextRuntime where
  return x = CR $ \c -> (x, c)
  m >>= k = CR $ \c ->
    let (a, c') = runContextRuntime m c
    in  runContextRuntime (k a) c'

newtype ContextRuntimeT m c1 c2 a =
  CRT { runContextRuntimeT :: c1 -> m (a, c2) }

instance (Functor m) => Functor (ContextRuntimeT m c1 c2) where
  fmap f (CRT cf) = CRT $ fmap (first f) . cf

instance (M.Monad m) => PMonad (ContextRuntimeT m) where
  return x = CRT $ \c -> M.return (x, c)
  m >>= k = CRT $ \c -> do
    (a, c') <- runContextRuntimeT m c
    runContextRuntimeT (k a) c'
```

```
liftCRT :: Monad m => m a -> ContextRuntimeT m c c a
liftCRT m = CRT $ \c -> m M.>>= (\a -> M.return (a, c))

cfToCr :: (k :▷ cs)
       => ContextF cs a -> ContextRuntime k k a
cfToCr cf = CR $ \k -> (evalC cf k, k)

cfToCrT :: (Monad m, k :▷ cs)
        => ContextF cs a -> ContextRuntimeT m k k a
cfToCrT cf = CRT $ \k -> M.return (evalC cf k, k)

inContextT :: (Monad m, k :▷ cs)
        => ContextF cs a -> ContextRuntimeT m k k a
inContextT = cfToCrT

cfToCrT' :: (Monad m) => ContextF cs a -> ContextRuntimeT m cs cs a
cfToCrT' cf = CRT $ \k -> M.return (evalC cf k, k)

updateC :: HUpdateAtTypeOrAppend e l c2
        => e -> ContextRuntime l c2 ()
updateC c = CR $ \c' -> ((), hUpdateAtTypeOrAppend c c')

updateCT :: (HUpdateAtTypeOrAppend e l c2, Monad m)
         => e -> ContextRuntimeT m l c2 ()
updateCT c = CRT $ \c' -> M.return ((), hUpdateAtTypeOrAppend c c')

pushC_PADL :: (HList cs, HUnion [h| c |] cs cr) => c -> ContextRuntime cs cr ()
pushC_PADL c = CR $ \cs -> ((), hUnion (c .*. hNil) cs)

emptyC :: ContextRuntime HNil HNil ()
emptyC = CR $ \c -> ((), c)

evalCR :: ContextRuntime HNil k a -> a
evalCR ca = fst . runContextRuntime ca $ hNil

evalCRT :: (Monad m)
        => ContextRuntimeT m HNil k a -> m a
evalCRT ca = runContextRuntimeT ca hNil M.>>= M.return . fst

execCR :: ContextRuntime HNil k a -> k
execCR ca = snd . runContextRuntime ca $ hNil

runCR :: ContextRuntime HNil k a -> (a, k)
runCR ca = runContextRuntime ca hNil
```

```
runCRT :: ContextRuntimeT m HNil k a -> m (a, k)
runCRT ca = runContextRuntimeT ca hNil

pushCT :: (Monad m) => c -> ContextRuntimeT m HNil c ()
pushCT c = CRT . const . M.return $ ((), c)

joinCRT :: (Monad m) => ContextRuntimeT m c c (ContextRuntimeT m c c a)
        -> ContextRuntimeT m c c a
joinCRT mma = mma >>= id

mcfToCrT :: (Functor m, Monad m) => (m a) :↓ c -> ContextRuntimeT m c c a
mcfToCrT = joinCRT . fmap liftCRT . cfToCrT

inContextM :: (Functor m, Monad m) => (m a) :↓ c -> ContextRuntimeT m c c a
inContextM = mcfToCrT
```

Context.Knowledge.HList

```
{-# LANGUAGE OverlappingInstances #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}
module Context.Knowledge.HList where

import Data.HList

import Context.Types

instance (HList c, HProject k c) => (:▷) k c where
  q = hProject
```

Context.Knowledge.Features

```
{-# LANGUAGE TypeFamilies, TypeOperators, QuasiQuotes #-}
module Context.Knowledge.Features where

import Data.Maybe
import Data.List
import Data.Function

import Control.Applicative

import Context.Utils
import Context.Implicit
```

```
type family FeatureType a :: *

data Feat a = a := (FeatureType a)
infixr 2 :=

data individual :> feature = individual :> (Feat feature)

π :: a -> f -> FeatureType f :+ [h| a :> f |]
π _ _ = mkC1 $ \(_ :> (_ := v)) -> v

extractFeature (_ :> _ := v) = v
```

Context.Knowledge.Relevance

```
{-# LANGUAGE TypeFamilies, TypeOperators, QuasiQuotes #-}
module Context.Knowledge.Relevance where

import Data.List

import Context.Prelude

class Relevant a where
  type RelevantK a :: *
  relevance :: a -> RelevantK a -> Double

sortC :: (Relevant c)
      => (a -> c) -> [a] -> [a] :+ [h| RelevantK c |]
sortC contextfn xs =
  let sortfn c x y = compare (relevance (contextfn x) c)
                             (relevance (contextfn y) c)
  in  ContextF (\c -> sortBy (sortfn . hOccurs $ c) xs)
```

Context.Prelude

```
module Context.Prelude (
    module Prelude,
    module Control.Applicative,
    module Data.HList,
    module Data.HList.TypeEqGeneric1,
    h,
    module Context.Implicit,
    module Context.Runtime
) where

import Prelude hiding (return, (>>=), (>>))
```

```
import Control.Applicative
import Data.HList ( HCons, hCons, HNil
                  , hNil, (:*:), (.*.)
                  , Fail, TypeNotFound, hOccurs)
import Data.HList.TypeEqGeneric1
import Context.Utils
import Context.Implicit
import Context.Runtime
import Context.Knowledge.HList
```