Prototyping a Prototyping Language

Case study: An Action-Semantics-based Approach for High-level Source-to-source Language Translation

by

Hseu-Ming Chen

A dissertation submitted in partial fulfillment of the requirements for the degree

of Doctor of Philosophy

Department of Computer Science

Graduate School of Arts and Science

New York University

May 1999

Dissertation Advisor

To my parents

Acknowledgment

I have been fortunate to carry out the research for this dissertation in the stimulating and supportive environment of the Griffin project. I would like to thank my research advisor, Professor Malcolm C. Harrison, Griffin's project leader, for his role in creating that environment as well as for his guidance, and careful reading of several drafts of my dissertation.

I would also like to express my appreciation to Professor Edmond Schonberg and Benjamin Goldberg for reading my dissertation and making helpful comments that greatly improved the presentation and enhanced the clarity of this work.

I am especially grateful to Deepak Goyal, for helping me in more ways than he can probably imagine; in particular, for his professional advice and answers to many of my questions on research and writing. I wish to express my appreciation to Zhe Yang, for helping me finalize some of the ideas appearing in this thesis and for being an authority on whose advice I could always count on. Edward Osinski's contribution was widespread, and his talent for penetrating the implementation problem of the translation of existential types was especially valuable. I would like to thank him for his friendship and assistance, making my experience here a very enriching one. Special thanks go to Allen Leung for sharing his insights into type theory, formal semantics, and numerous useful reference pointers.

In the Computer Science Department as a whole, I would like to thank all my friends at NYU in particular Professor Marsha Berger, who has always provided

encouragement through the long process of writing a dissertation; Professor Vijay Karamcheti and Professor Robert Dewar for serving on my committee; Professor Robert Paige, Professor Richard Cole, Ting-jen Yen, Fangzhe Chang, David Tanzer, and Anina Karmen for all their support and friendship.

Most of all, my gratitude goes to my wife, Wan-Pi Yang, for her faith and support at all times and to my children, Kevin and Vivian for always filling my life with joy.

Abstract

The development of a prototyping language should follow the usual software-engineering methodology: starting with an evolvable, easily modifiable, working prototype of the proposed language. Rather than committing to the development of a mammoth compiler at the outset, we can design a translator from the prototyping language to another high-level language as a viable alternative. From a software-engineering point of view, the advantages of the translator approach are its shorter development cycle and lessened maintenance burden.

In prototyping language design, there are often innovative cutting-edge features which may not be well-understood. It is inevitable that numerous experimentations and revisions will be made to the current design, and hence supporting evolvability and modifiability is critical in the translator design.

In this dissertation we present an action-semantics-based framework for high-level source-to-source language translation. *Action semantics* is a form of denotational semantics that is based on abstract semantic algebra rather than Scott domain and λ-notation. More specifically, this model not only provides a formal semantics definition for the source language and sets guidelines for implementations as well as migration, but also facilitates mathematical reasoning and a correctness proof of the entire translation process. The translation is geared primarily towards readability, maintainability, and type-preserving target programs, only secondarily towards reasonable efficiency.

We have acquired a collection of techniques for the translation of certain non-trivial high-level features of prototyping languages and declarative languages into efficient procedural constructs in imperative languages like Ada95, while using the

abstraction mechanism of the target languages to maximize the readability of the target programs. In particular, we translate Griffin existential types into Ada95 using its object-oriented features, based on coercion calculus. This translation is actually more general, in that one can add existential types to a language (with modicum of extra syntax) supporting object-oriented paradigm without augmenting its type system, through intra-language transformation. We also present a type-preserving translation of closures which allows us to drop the whole-program-transformation requirement.

Contents

List of Figures

Chapter 1

Introduction

The growing size and complexity of software systems make it increasingly difficult — if not impossible — to obtain an exact, complete requirements specification from software clients or end-users. Careful requirement analysis along with systematic reviews of the requirements help to reduce the uncertainty about what the system should do. However, there is no real substitute for trying out a requirement before committing to it. This is possible if a software prototype of the system to be developed is available. System prototypes allow users to experiment with requirements and to examine how the system supports their work. Prototyping is therefore a means of requirement validation, which permits clients to discover requirement errors or omissions early in the software process.

The main advantages of developing a prototype early in the software process are:

- Prototypes serve as a basis for writing the specification for a production quality system.

- Prototypes are used for experimental purposes and for gaining practical experience.
- Prototypes are used to clarify any relevant specification or development problems.
- Prototypes serve as a basis for discussion and as aids to decision making.
- Prototypes help in identifying errors and omissions in requirements.
- A working, albeit limited, system is available quickly to demonstrate the feasibility and usefulness of the application.

Experiments [81] have shown that prototyping reduces the number of problems with the requirement specification and the overall development cost.

Prototyping languages (also referred to as *modelling languages*, *executable specification languages*, or *problem-oriented implementation languages*) are programming languages appropriate for constructing executable prototypes of development-intensive software. The most important property of a prototyping language is its ability to explore the problem and solution space at low cost during early software development stages. This is usually intepreted as the ability to express the essential content of an algorithm while leaving out unnecessary details. This in turn requires that prototyping languages possess powerful data processing capabilities, which simplify program development by alleviating many of the problems related to storage allocation and management. Such language systems often include many facilities which would normally be built from more primitive constructs in other languages.

In order for prototyping to be reasonable, the cost of experimentation has to be low. This cost depends on two factors: the cost of implementing an initial version

of a software system and the cost of evolving it. Thus the time to write the code for an experiment must be short enough that the code can be discarded if the idea fails to produce the desired result.

The costs of experimentation depend on the suitability of the applied programming language and programming environment for fast implementation, modification, and reuse of code. The programming languages for prototyping should be expressive enough to offer a vast amount of functionality in little code, and support modifiability and extensibility.

1.1 Prototyping a high-level programming language

In the 1950s, it was widely believed that efficient programs could be crafted only by hand, using low-level languages. By a low-level language we mean either a machine language, or an assembly language, which is just is a mnemonic variant of a machine language in which names take the place of the actual codes for machine operations, values, and storage locations. Unfortunately, programs in machine language or assembly language are unintelligible. In early 1960s, John Backus of IBM advocated his "Speedcoding" idea after he observed that it is more expensive to design and debug a program than to run it. Henceforth we reserve the term "programming languages" for *high-level* languages, where high-level connotes greater distance from machines. High-level languages allow programs to be specified more in terms of algorithmic concepts without exposing excessive implementation details. The high cost of manually creating machine or assembly code was a prime motivation for the development of high-level programming languages.

Any notation other than machine language cannot be run directly on a machine;

it must be translated into a form the machine can understand. A translator from a programming language into machine or assembly code is called a *compiler* [4, 35, 78]. Hereafter we use the term *translator* to refer to the programs that translate code written in a high-level language into another high-level language.

The executable version of a program written in language $\mathcal{L}_S$ can be obtained either (1) by writing a $\mathcal{L}_S$ compiler from scratch, or (2) by writing a language translator from source language $\mathcal{L}_S$ to target language $\mathcal{L}_T$, where a production quality compiler is already available for $\mathcal{L}_T$. Executing the target program on a computer captures the semantics of the source program. It is more cost-effective to develop a translator than a compiler: for the same reasons that it is easier for one to write programs in a higher-level language than in assembly, it is easier to write a program which outputs programs in a higher-level language than in assembly. The major advantages of the translator approach are its shorter development cycle and lessened maintenance burden. If the translator itself also serves as a formal semantic description of the source language, this approach conforms to a desirable methodology of language design: express the design as a formal specification, and use this to test and refine the design, before freezing the design or becoming committed to constructing a compiler.

For many applications, researchers and practitioners in recent years have developed different domain specific languages, or DSLs, which are tailored to particular application domains. With an appropriate DSL, you can develop complete application programs for a domain more quickly and more effectively than with a general purpose language. Ideally, a well-designed DSL captures precisely the semantics of an application domain, no more and no less.

A partial list of domains for which DSLs have been created includes lexing, parsing distributed computing, scheduling, parallel computing, logic, modeling, simulation, graphical user interfaces (GUI builder), symbolic computing, cad, cam, robotics, hardware description, silicon layout, pattern matching, graphics, animation, computer music, databases, and security. By the nature of DSLs there will be many of these and development ease is important. Our approach, which really attacks a more difficult problem (Griffin being a broad-spectrum language) than most DDLs will present, can be useful in those cases also.

1.2 Motivation

The development of a prototyping language should also follow the usual software-engineering methodology: starting with an evolvable prototype which is an easily modifiable and extensible working model of the proposed language. This miniature prototype should provide a formal description of the semantics of the language's syntactic constructs. The process of precisely defining the meaning of the syntactic constructs of a programming language can reveal all kinds of subtleties of which it is important to be aware.

Rather than committing to the development of a compiler at the outset, a translator from the prototyping language being designed to another high-level language is a viable alternative. From a software-engineering point of view, the advantages of the translator approach are its shorter development cycle and lessened maintenance burden. A translator is a fast and inexpensive testbed of the language design, and can take advantages of existing tools for the target language. Since the target code of the translator will be eventually processed by an existing compiler, we automat-

ically benefit from various compiler optimizations implemented in the compiler for the target language.

Like the construction of a compiler, the structure of a translator requires careful design as well. Program translation is related to exploring the semantic structure of the source program to produce a target program with matching semantics. There are many commercial or free program translators (or "converters" as they are sometimes called) available; among them, Fortran to C [51, 31] and Pascal to C translators have been widely used. For source-to-source program translation, it is often the case that source programs are written in a higher-level language which is more expressive than the target language, where translation is done mainly for efficiency purposes. However, the translations [56, 31] are usually carried out in a brute-force way by writing an ad hoc program to achieve the translation task, as a consequence of which we have to face the usual software-engineering problems such as readability and maintenance of the unwieldy translation program. Alternatively we can base the translation process on a formal semantics framework. The semantic metalanguage used in the description can be viewed as the operational understanding of the source language, whereas the formal semantics notation can serve as an intermediate language for an translator, interpreter, or compiler. As compiler optimization techniques improve, so does the efficiency of the programs written in the target language. Therefore, we consider the translator approach pragmatically promising.

1.3 Contribution

In prototyping language design, there are often innovative cutting-edge features which may not be well-understood. As a consequence, it is inevitable that numerous experimentations and revisions will be made to the current design. For speedy development, it is crucial to have a translator for the prototyping language which is easily modifiable. Unfortunately, it is difficult to structure the translation process plausibly; in most cases language translators appear amorphous and hard to follow, forcing us to face the usual software-engineering concerns such as readability and maintenance of the unwieldy translation program.

In this dissertation we present a formal-semantics-based model for high-level source-to-source language translation, which illustrates the applicability of formal semantics as a tool in computer science. More specifically, this model not only provides a formal semantics definition for the source language and sets guidelines for implementations as well as migration, but also facilitates mathematical reasoning and a correctness proof of the entire translation process.

We claim our approach to developing language translators is superior to conventional methods. Our primary considerations in the design of the language translator are, in decreasing order of importance: rapid development of a language translator for fast prototyping (exploratory programming), making provision of formal semantic description for the source language, a more abstract and more readable translator that readily accommodates changes to allow the translator to evolve with the language. The formal semantics based two-level model for structuring language translators is suitable for reasoning about the semantics of the source language from the high-level semantic specification, and at the same time produces

reasonably efficient target code according to the low-level specification.

This model employs the framework of action semantics to decouple the analysis of the source program and the generation of the target program. *Action semantics* is a practical framework for modular semantics of programming languages. It is a member of the functional semantics family characterized by the use of compositional valuation functions. Such decoupling eases the retargeting of the translation to other languages. The type inference/checking scheme for the source language is completely parameterizable with respect to the source language. Our implementation demonstrates that the model is essentially a collection of reusable components which allows other language translators to be built upon it, thereby increasing the productivity of translation program writers.

Most programmers understand programming languages in terms of basic concepts such as control flow, binding, store update and parameter passing. Unfortunately, formal specifications often obscure these notions to the point that the reader must invest considerable effort to determine, for example, how parameters are actually passed, or whether a language follows static or dynamic scoping. It is frustrating that some of the most fundamental concepts of the programming language are the hardest to understand in a formal definition. The original motivation for the development of action semantics was dissatisfaction with pragmatic aspects of denotational semantics [64]. The primary aim of action semantics is to allow formal, yet more accessible semantic descriptions of realistic programming languages by using concepts familiar to most programmers. This is exactly what some semanticists have been advocating over the years: *making computer science concepts explicit in the formal description.*

Unlike some λ-calculus based translation programs that are difficult to understand, both our action-semantics-based semantic description and the translation program are highly intelligible. Our approach also lessens the burden for future maintenance as compared to an approach yielding an unwieldy translator. The semantic description of the source language is directly executable so it can be used to automatically obtain a static semantics analyzer (in analogy to the language's syntax, context-free grammars have similar features).

We also detail the translation of certain non-trivial high-level features of prototyping languages and declarative languages into efficient procedural constructs in imperative languages like Ada95, while using the abstraction mechanism of the target languages to maximize the readability of the target programs. In particular, we translate Griffin existential types into Ada95 using its object-oriented features, based on coercion calculus [60, 41, 40]. This translation is actually more general, in that one can add existential types to a language (with a modicum of extra syntax) supporting the object-oriented paradigm without augmenting its type system, through intra-language transformation. We also present a type-preserving translation of closures which allows us to drop the whole-program-transformation requirement.

In the interest of readability, we would like to keep the generated code for operator expressions as *parenthesis-free* as possible. A straightforward algorithm is presented to minimize the the number of generated parentheses as long as the semantics is preserved.

Lastly, a monadic-style multi-level pretty-printer is implemented to generate nicely formatted textual representations of Ada programs from Ada abstract syntax

trees.

1.4 Overview of the dissertation

In Chapter 2 we review various language translation techniques, and provide a brief survey of several formal semantics frameworks and the evolution of action semantics. Chapter 3 depicts a semantics-based language translator and the advantages of such an approach. The translator G2A that we experiment with in this model is described in Chapter 4. Chapter 5 gives a brief introduction to the source language Griffin, and explains how the static analysis is performed according to the macrosemantics specification. In Chapter 6 we present a quick overview of the target language Ada95. Chapter 7 details the generation of the target code with respect to the macrosemantics specification and translations of certain features in Griffin that do not have a straightforward translation into Ada95. Finally, in Chapter 8 we discuss some of the overall design choices and conclude with directions for future work. The Appendix contains the Griffin grammar specification and macrosemantic equations written in TML for Griffin generators.

Chapter 2

Background

Software prototyping reduce the risks of incomplete and erroneous requirements definitions. Prototyping methods base on the idea that many misunderstanding between developers and clients as well as many errors can be eliminated during the requirements definition process if working software prototypes of the planned applications are built and examined.

The history of software prototyping can be viewed as the constant quest for better control of costs, quality, and development time. The prototyping activities follow up and document trial applications of the proposed technologies, demonstrate the scope, applicability, benefits and costs of the proposed solution, and investigate possible synergies among them. Software prototyping is a specific strategy for improving requirements definitions wherein user needs are extracted, presented and successively refined by building a working model of the ultimate system quickly and in context. Prototyping languages are programming languages appropriate for constructing executable prototypes of development-intensive software. In most cases, prototyping languages are used today to implement prototypes. The simple

reason for this is that, if the prototype requires modification, a prototyping language will help to carry out the modification efficiently. It is quite straightforward to think of a prototyping language as an implementation language for the anticipated product. But there is a second possibility: that of using a prototyping language to describe the design process itself.

A number of approaches to language translation are especially notable and reviewed in more detail here. Section 2.4 describes various formal semantic frameworks.

2.1 Software prototyping and prototyping languages

Software prototypes are designed to clarify certain aspects of the system for both computer experts and end-users. For this purpose, the two groups construct a program, analyze it, evaluate it in operation, and modify it. Prototyping languages are designed to facilitate this trial-and-error process.

A software prototype is an easily modifiable and extensible working model of a proposed system, not necessarily representative of the complete system, which provides users of the application with an operational representation of key parts of the system before implementation. In other words, it is an easily built, readily modifiable, ultimately extensible, partially specified, working model of the primary aspects of a proposed system.

Software prototyping is an approach based on an evolutionary view of software development and having an impact on the development process as a whole. Software prototyping involves producing early working versions (prototypes) of the future application system and experimenting with them. It provides a communication

basis for discussion among all the groups involved in the development process, especially between the end-users and developers. Software prototyping also enables developers to adopt an approach to software construction based on experiment and experience.

Prototyping in the software development process may involve "throw-away" prototyping in which a prototype is developed to understand the system requirements, or evolutionary prototyping in which a prototype evolves through a number of versions into the final system. Prototyping techniques include the use of executable specification languages or very-high-level languages, and prototype construction from reusable components. Rapid development is important for prototype systems. To deliver a prototype quickly, you may have to leave out some system functionality or relax non-functional constraints such as response time and reliability.

A class of programming languages which have been proposed as prototyping languages are so-called *multi-paradigm* or *wide-spectrum* programming languages. A wide-spectrum language is a programming language which incorporates a number of paradigms. As an alternative to using a wide-spectrum language, one can use a *mixed-language* approach to prototype development. Different components of the system may be programmed in different languages and a communication framework established among the components. The advantage of a mixed-language approach is that the most appropriate language can be chosen for each logical part of the application, thus speeding up prototype development. The disadvantage is that it may be difficult to establish a communication framework which will allow multiple languages to communicate. The entities used in the different languages may be very diverse. Consequently, lengthy code sections may be needed to transport an

entity from one language to another.

It is important that the code written in a programming language have a *high semantic density*, which means that little code has to be written to implement an algorithm. An example of a programming language with high semantic density is APL. APL makes it possible to formulate complex algorithms in a few lines – unfortunately such programs are often not comprehensible to anybody but the implementor. Because of its semantic density APL is excellently suited for prototyping of short algorithms, but less appropriate for exploratory programming of large software systems.

Programs written in a prototyping language should be compact. By this, we mean, for example, that the formulation of a graph theory problem should not be much longer than the mathematical formulation itself. One example of an appropriate language in this case would be the mathematically oriented prototyping language SETL [77].

Prototyping languages are of interest for the purposes of both specification and implementation. Both aspects highlight the benefits prototyping languages offer for prototype construction.

2.2 Source-to-source language translation

There are many reasons for being interested in a high-level translation, instead of direct compilation to machine code. The usefulness of program translation stems from practical considerations such as reducing development time and increasing software reliability. Additionally, if the translator itself also serves as a formal description of the source language under design, then the development process as a

whole supports a methodology of language design that we advocate: use a formal description to test and refine the design, before committing to the construction of a substantially more complicated compiler.

As compiler technology for the target language improves, programs translated into this target language benefit automatically. For example, C is a commonly chosen target language. Since a great deal of effort has been made to optimize C compilers, execution of the translated source inherits these improvements as well.

Modern prototyping languages permit rapid software development at the expense of execution-time performance. If $\mathcal{L}_H$ is a prototyping language, a $\mathcal{L}_H$ to $\mathcal{L}_L$ translator where $\mathcal{L}_L$ is a production language achieves fast prototyping from $\mathcal{L}_H$ to $\mathcal{L}_L$. For example, in the Griffin project developed at NYU, Griffin is the prototyping language and Ada95 the production language. G2A is the translator that transforms the initial prototype expressed in Griffin into production quality code in Ada95.

Apart from the major advantages like reducing the development cycle and providing solid formal semantic basis (if based on a formal semantics model) mentioned before, there are many other real world applications of program translation. Let $\mathcal{L}_S$ and $\mathcal{L}_T$ denote the source language and target language respectively. As an example, program translation is often used to promote language *interoperability* (especially in a transition or development stage of the source language). One can portably mix $\mathcal{L}_S$ and $\mathcal{L}_T$ programs (source-level integration) and make an existing base of well-tested $\mathcal{L}_S$ code available to $\mathcal{L}_T$ programmers. Translating code from one language to another is also a way of preserving the investment made in the ported part of the system. For instance, there is a large body of well tested Fortran

code for carrying out a wide variety of useful calculations and it is appealing to exploit some of this Fortran code in a C environment.

Some existing compilers have the ability to call external functions written in another language (usually C). Unfortunately, due to differences in memory models and type systems make the so-called *foreign function interfaces* approach awkward to use, limited in functionality, and even type-unsafe. Worst yet, it is infeasible in some situations because of the inefficiency resulting from inter-language on-the-fly data conversion and control transfer. However, a language translator serves the same purposes well without suffering the drawbacks mentioned above.

The language translator approach is also superior in the case when it is useful to run a well-tested $\mathcal{L}_S$ program in an environment that only has an $\mathcal{L}_T$ compiler but not an $\mathcal{L}_S$ compiler; or when there exist many useful tools for $\mathcal{L}_T$ but not $\mathcal{L}_S$, source-level integration of $\mathcal{L}_S$ with $\mathcal{L}_T$ makes those tools available to $\mathcal{L}_S$ code.

Program transformations are concerned with semantics–based analysis and manipulation of programs for improving their efficiency. Certain programming languages are more amenable to certain kinds of transformations. For example, *partial evaluation* [50] (or program specialization), a widely advocated program optimization technique, is more effective on functional programming languages. Partial evaluation provides a unifying paradigm for a broad spectrum of activities in program optimization, interpretation, compiling and other forms of program generation, and even generation of automatic program generators. Although partial evaluation has been the subject of rapidly increasing activity over the past few years, it is still not mature enough to be directly applicable to imperative programming languages. However, if we have a translator from an imperative language to a functional lan-

guage for which there exists a partial evaluator, then the efficiency of programs in the imperative language can be improved by partially evaluating the translated version of these programs.

Due to the expressiveness difference between $\mathcal{L}_S$ and $\mathcal{L}_T$, some languages are simply more appropriate for certain purposes. For instance, it is impossible or harder to express in Fortran77 than in C such features as storage management, character operations, array of functions, heterogeneous data structures and calls that depend on the operating system.

From time to time, programmers may need some features in $\mathcal{L}_T$ but still be able to use existing code in $\mathcal{L}_S$; or some programmers simply prefer one language to another and they write components in the source language that can be integrated into larger target language based systems.

Another important use of program translation may be to make successively high-level specification languages *computationally transparent*. A language is computationally transparent if it is amenable to algorithm analysis [19]. If the source language $\mathcal{L}_S$ is not computationally transparent (or worse yet, $\mathcal{L}_S$ could be a very high-level specification which is not directly executable), one way to analyze the time complexity of programs in $\mathcal{L}_S$ is to write a translator from $\mathcal{L}_S$ to $\mathcal{L}_T$ where $\mathcal{L}_T$ is a computationally transparent language. Assume there is a hierarchy of languages $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$, where $\mathcal{L}_i$ is higher-level than $\mathcal{L}_{i-1}$ and $\mathcal{L}_1$ is computationally transparent. If there also exist language translators from $\mathcal{L}_i$ to $\mathcal{L}_{i-1}$ for $2 \leq i \leq n$, all languages in this hierarchy become computationally transparent.

2.3 Approaches to language translation

Translating one programming language to another, rather than compiling to specific machine's instruction set, is a very old idea. There are many real world applications of language translation we already mentioned in Section 2.2. The translation approach has yet a few more advantages over writing a compiler. First, by emitting code written in an intermediate language rather than machine code, a translator is extremely portable. Second, the intermediate language provides a common meeting point for all languages and thus facilitates the construction of program written in multiple languages. Finally, the compiler of the intermediate language can simplify the individual language compiler by providing language-independent optimization.

Various approaches related to language translation will be discussed in the following sections.

2.3.1 Technical prerequisites

If we define:

$\mathcal{L}_S$ = source language

$\mathcal{L}_T$ = target language

$\mathcal{L}_I$ = implementation language of the translator

$\mathcal{P}_S$ = source program written in $\mathcal{L}_S$

$\mathcal{P}_T$ = target program written in $\mathcal{L}_T$

Φ = translation program written in $\mathcal{L}_I$

$\mathcal{I}$ = input domain

$d_1, \ \ldots, d_n \ \in \ \mathcal{I}$

$\mathcal{O}$ = output domain

$output \ \in \ \mathcal{O}$

then the translation process in general can be described as follows:

$$\begin{aligned} output &= [\![\mathcal{P}_S]\!]_{\mathcal{L}_S} \ [d_1, \ldots, d_n] \\ &= [\![\Phi]\!]_{\mathcal{L}_I} \ [\mathcal{P}_S] \ [d_1, \ldots, d_n] \\ &= [\![\mathcal{P}_T]\!]_{\mathcal{L}_T} \ [d_1, \ldots, d_n] \end{aligned}$$

where the emphatic syntax brackets, $[\![\ ]\!]$, delimit the syntactic entities and $[\![\mathcal{P}]\!]_{\mathcal{L}}$ gives the denotation (or meaning) of program $\mathcal{P}$ in language $\mathcal{L}$.

Fritz Henglein mentioned that high-level translation between statically and dynamically typed languages is a notoriously tricky business, even for semantically and syntactically — seemingly — "compatible" language families such as Scheme and ML [40]. One goal of our translation process is to preserve the appearance of the source program in the target program in addition to preserving the semantic features of the original program whenever possible. This facilitates migration of programs from $\mathcal{L}_S$ to $\mathcal{L}_T$. However, due to the "semantic gap" [13] between the source and target languages this goal is not always feasible in practice. For example,

- If the source language is imperative and target language is pure applicative, then every program point in the target code must contain extra information about the "current configuration".

- If the source program uses continuation for control flow but the target language does not support it, then every program point in the target code must contain extra information about the "current continuation".

- If the exception mechanism is present in the source language but not in the target language, then every program point in the target code must contain extra information about the "current exception handler".

- There are some forms of static constraint in the source language which have no natural counterparts in the target language. For example, the class constraints imposed on Griffin functions will be translated into arity-raised Ada95 functions.

All the examples show that the source language is "more expressive" than the target language (or at least the target language's expressiveness does not cover that of the source language) as defined by Felleisen in [32]. In each case, non-local translation of the program is required.

It is almost inevitable that we will run into situations like these mentioned above in the translation process, and we may sometimes have to settle for target code that is not very similar to the source code but has the same dynamic computational effects.

Just as no one ever attempts to prove the correctness of a real compiler such as `gcc`, we only informally claim the partial correctness of G2A. That is, either $[\![\mathcal{P}_S]\!]_{\mathcal{L}_S}\ [d_1, \ldots, d_n]$ and $[\![\mathcal{P}_T]\!]_{\mathcal{L}_T}\ [d_1, \ldots, d_n]$ both evaluate to the same value, or neither evaluation is defined:

$$\forall\, d_1, \ldots, d_n\ \in\ \mathcal{I}$$

$$(\,[\![\mathcal{P}_S]\!]_{\mathcal{L}_S}\ [d_1,\ldots,d_n]\ \downarrow)\ \&\ (\,[\![\mathcal{P}_T]\!]_{\mathcal{L}_T}\ [d_1,\ldots,d_n]\ \downarrow) \quad \Rightarrow$$
$$[\![\mathcal{P}_S]\!]_{\mathcal{L}_S}\ [d_1,\ldots,d_n]\ =\ [\![\mathcal{P}_T]\!]_{\mathcal{L}_T}\ [d_1,\ldots,d_n]$$

Ideally what we would like to have is if the source program terminates, so does the target program and the outputs from them are identical:

$$\forall\, d_1,\ldots,d_n\ \in\ \mathcal{I}$$
$$[\![\mathcal{P}_S]\!]_{\mathcal{L}_S}\ [d_1,\ldots,d_n]\ \downarrow\ \Rightarrow$$
$$(\,[\![\mathcal{P}_T]\!]_{\mathcal{L}_T}\ [d_1,\ldots,d_n]\ \downarrow)\ \&\ (\,[\![\mathcal{P}_S]\!]_{\mathcal{L}_S}\ [d_1,\ldots,d_n]\ =\ [\![\mathcal{P}_T]\!]_{\mathcal{L}_T}\ [d_1,\ldots,d_n]\,)$$

where $\mathcal{P}(i)\downarrow$ means that $\mathcal{P}(i)$ is defined (i.e., the computation terminates).

The notation $[\![\_]\!]_{\_}$ is overloaded here. $[\![\mathcal{P}_S]\!]_{\mathcal{L}_S}$ denotes the semantics of the program $\mathcal{P}_S$ written in language $\mathcal{L}_S$. Henceforth, we will also use $[\![\mathbb{S}]\!]_{\texttt{cat}}$ to denote the meaning of the syntactic construct $\mathbb{S}$ which belongs to the syntactic category `cat`. For instance, $[\![\mathbb{E}]\!]_{\texttt{exp}}$ and $[\![\mathbb{D}]\!]_{\texttt{decl}}$ are denotations of the syntactic expression $\mathbb{E}$ and declaration $\mathbb{D}$, respectively. The meaning that we have in mind can always be determined from the context.

2.3.2 Ad hoc hand-written language translators

There are several general approaches that a language translator writer can adopt to implement a translator. The oldest is by writing an ad hoc translation program to carry out the translation task in a brute force way; consequences of this are the usual software-engineering problems such as readability and maintenance of the unwieldy translation program.

A common practice in writing translators is to rely on *boot-strapping* and step-wise refinement. One form of bootstrapping builds up a translator for larger and

larger subsets of the source language. Recall $\mathcal{L}_S$ and $\mathcal{L}_T$ denote the source and target language respectively. As a first step we might write a small translator that translate a subset $\mathcal{L}_{S_1}$ of $\mathcal{L}_S$ into the target code; that is, a translator $\Phi_T^{S_1}$. We then use the subset $\mathcal{L}_{S_1}$ to further derive an executable translator Φ_T^S for $\mathcal{L}_S$.

In a real sense, a translator is just a program. The environment in which this program is developed can affect how quickly and reliably the translator is implemented. The language in which the translator is implemented is equally important. Using the bootstrapping technique, running the translator helps debug the translator at each stage.

Unfortunately bootstrapping is not good enough since it does not explicitly give the formal semantics of all language constructs. Worse yet, it also introduces the *meta-circularity* problem in the source language definition.

Ease of maintenance and readability depend on the provided structure and documentation of the translator. Language translators should be engineered like other software products rather than be developed in an ad hoc manner in which a systematic software development process is lacking.

2.3.3 λ-calculus based hand-written language translators

Some researchers base the language translation tasks on λ-calculus [12, 23, 72, 76, 83, 86, 88]. Usually they express control flow with continuations, or continuation-passing style [5], or variations of them. Their translations usually take a few steps, with each step serving a different purpose. However, it clearly is unreasonable to expect the readers of the semantic description of any programming language to be expert in the idiosyncrasies of standard denotational semantics description

techniques. Even if the readers are familiar with continuations in denotational semantics, it is inevitable that over-specification occurs in parallel syntatic constructs due to the fact that λ-calculus is inherently sequential [8]. Furthermore, formal specifications in λ-calculus are difficult to create accurately, to modify, and to extend.

2.3.4 Program Transformation and Partial Evaluation

Programmers and algorithm designers have been using the idea of prototyping all the time. The discovery of an efficient algorithm often starts with a concise mathematical specification or conceptual understanding of the problem—a high-level prototype—which, through rounds of refinements, finally evolves into an efficient program. Such refinements are based on observations of the prototype: by exploiting such observations, applying certain schemas, programs are tuned to be more efficient. Over the years, repeated observations and common schemas are abstracted to become semantic-preserving transformation rules which apply to large class of situations. *Program transformation* techniques, concerned with semantics–based analysis and manipulation of programs for improving their efficiency, are developed to automate the program derivation process by mechanically applying such rules.

Burstall and Darlington established the area of transformational programming [24]. They studied the basic transformation in a functional setting: for example, the *unfolding* transformation replaces a call-site by the instantiated body expression of the callee, and its inverse *folding* transformation generalizes expressions to get function definitions. Robert Paige [71] proposed the finite difference transformation based on the observation that the programmers manually reduce the cost

of computing an expression in a loop by maintaining its value using a variable, and update it incrementally. The finite difference technique has been used to justify and discover several very efficient algorithms. The fusion technique (or "deforestation") by Wadler eliminates intermediate data structures by combining the generation process and the consuming process of the data structure. Recently Zhenjiang Hu [42] developed an inverse transformation called *diffusion* operation that, by introducing more intermediate data structures, breaking programs into small functions that can be easily turned into parallel programs.

Partial evaluation [50] (or *program specialization*, *mixed computation*) is a general and probably the most automated program transformation technique. Partial evaluation provides a unifying paradigm for a broad spectrum of activities in program optimization, interpretation, compiling and other forms of program generation, and even generation of automatic program generators. The idea of partial evaluation is based on the observation that, often, some parameters (called static inputs) to a program change less frequently than the others (called dynamic inputs); when the static inputs are known, one can first carry out computations that only depend on them and generate a specialized (and hopefully simplified) *residual program*, which, taking the dynamic inputs, generates the final result. The residual program often runs much faster than the general program, and the cost of performing the partial evaluation is quickly amortized over several runs of the residual program. In fact, this pattern of specializing general-purpose program is not unfamiliar to many programmers. Most programmers are constantly deciding whether to write a general but less efficient program, or several special-case programs that runs efficiently; experienced programmers often end up writing template programs

that they will instantiate to special cases manually. Partial evaluation accelerates this process and makes it less error-prone.

Partial evaluation are often used to automatically and correctly generate language translators from *definitional interpreters*. A definitional interpreter directly gives an operational semantics for the interpreted language. It is often much easier to write an interpreter than to write the corresponding compiler (or language translator in general). One reason is that an interpreter implementor thinks only of the execution time, whereas a compiler implementor must distinguish compile-time from run-time in order to generate code to achieve a desired effect at run-time: compile-time computation should be performed, while run-time computation should be captured by the generated code. A partial evaluator can figure out the binding-time of each computation based on which inputs are available, thereby automatically splitting the single execution time of an interpreter into the compile-time and the run-time of a compiler. The idea was initially summarized by Futamura in his three projections [36]:

1. Specializing the interpreter with a particular program gives a compiled program.

2. Specializing the partial evaluator with an interpreter gives a corresponding compiler.

3. Specializing the partial evaluator with the partial evaluator gives a compiler generator, i.e., a transformer that turns interpreter to compiler.

These so-called Futamura projections are discovered in early 70s, but there had been little progress of the second Futamura projection until 1985, when Neil Jones

et al. [49] used a separate binding-time analysis to achieve self-application. Since then, the area of partial evaluation flourished with new advances in the fundamental theory, in the techniques, and in applications.

A feasible alternative to produce a language translator is to perform partial evaluation on an interpreter from the source language to the target language. However, apart from the promising developments in recent years, partial evaluation is still inadequate to be used to structure large language translation task:

- It is hard for a user to control the specialization process, especially when one does not have enough knowledge about how the partial evaluator works.
- Lots of limits in the source program will be inherited in the residual code, preventing the interpretative overhead of the original program from being completely removed.

There has been a sizeable amount of work devoted to address these problems. Progress has been made, and it seems that partial evaluation is coming close to be a pratical component of a language translator.

2.4 Formal semantics

Although most programmers rely on informal specifications of programming languages, these definitions are often vague, incomplete, and even erroneous. English just does not lend itself to precise and unambiguous definitions; it is ill-suited for use by an implementor, or by someone who wants to formulate laws for equivalence of programs, or by a programmer who wants to design programs with mathematical rigour.

The importance of applying formal semantics to programming language development is widely recognized. It serves as a basis for understanding and reasoning about how programs behave. Not only is a mathematical model useful for various kinds of analysis and verification, but also, at a more fundamental level, because simply the activity of trying to define the meaning of program constructions precisely can reveal all kinds of subtleties of which it is important to be aware. By using appropriate formal semantics techniques, the definition of a programming language or meaning of a program fragment is more accessible and succinct.

Formal semantics is important to programming languages in many ways:

- The experience of writing a formal description should assist the language designer to uncover inconsistencies and ambiguities in the design process. Only after this design-specification-testing sequence iteration has converged should the major effort of constructing a compiler be initiated.

- In practice, language design and implementation are often interleaved and iterated. A comprehensive description of the language is needed to convey the intentions of the language designers to the implementors.

- It is also needed for setting a definitive standard for implementations, so that programs can be transported between different implementations that conform to the standard, without modification. In short, it aids in the portability issue.

- Programmers need a description of any new language in order to relate it to previously known ones, and to understand it in terms of already familiar concepts.

- The programmer also needs a description as a basis for reasoning about the correctness of particular programs in relation to their specification, and for justifying program translations.
- Theoreticians can obtain new insight into the general nature of programming languages by developing descriptions of them.

2.4.1 Classical approaches

Historically the semantics of programming languages is often viewed as consisting of three approaches:

Denotational semantics is a technique for defining the meaning of programming languages pioneered by Christopher Strachey and furnished with a mathematical foundation by Dana Scott. It was called "mathematical semantics" at one time because it used the more abstract mathematical concepts of complete partial orders, continuous functions and least fixed points.

Denotational semantics is an attractive formalism for describing the semantics of programming languages. It is less abstract than axiomatic semantics, but on the other hand it is free of the excessive implementation details of operational semantics, allowing the reader to concentrate on understanding the abstract meaning of the constructs of the described language rather than their implementation.

Operational semantics describes the meaning of a programming language by specifying how it executes on an abstract machine. The method advocated by Gordon Plotkin in his lectures [74] at Aarhus University on "structural

operational semantics" (also called *natural semantics*), in which evaluation and execution relations are specified by rules in a way directed by the syntax is probably the most prevalent one nowadays.

Axiomatic semantics , the foundations of which depend on predicate logic (Hoare Logic in particular) tries to fix the meaning of a programming construct by giving proof rules for it within a program logic. In an axiomatic semantics model, axioms and deduction rules from mathematical logic are specified for each construct in a language. These allow assertions to be made about what is true after the execution of a language construct relative to what was true before (provided that the execution terminates). The nature of the assertions depends on the kinds of properties described by the axioms and rules.

Its main advantage is that the assertions it involves can be expressed in the programming language itself (sometimes with a modicum of extra apparatus, such as quantifiers), and there is a minimum of new notation. This is significant, as it is to be hoped that the proof of a program's correctness will become the responsibility of the programmers, and it is unreasonable to expect them to do the programming and the proving in two different languages.

However, each approach has its drawbacks and some of them prevent the use of it in automatic compiler construction or language translation in particular.

Denotational semantic descriptions expressed in λ-notation leave a lot to be desired from a software engineering point of view. The fundamental concepts in terms of which we understand, use and design programming languages are not expressed directly in the semantic equations; instead the concepts are encoded

in terms of function abstraction and application. For example, the fundamental concepts of binding, look-up, storing, retrieving, sequencing and argument passing are all encoded (ultimately) in terms of function application! Not surprisingly, such semantic descriptions are tedious to write, error-prone and difficult to read [52].

Denotational descriptions can be difficult to modify to accommodate certain new language features. For example, if we add a unconditional jump (goto) command to a language whose semantic description is in "direct" style, it would not simply be a matter of adding one semantic equation: the entire set of semantic equations would have to be converted from the direct style to the "continuation" style, since the former cannot easily accommodate jumps.

Another problem with λ-notation is that parallel activities cannot be satisfactorily described with this approach [8], i.e., the λ-calculus can not *directly* express parallelism (of course, because λ-calculus has been shown to be equivalent to Turing machines, one can indirectly express parallelism with a function which simulates the execution of a program in a lock-step way to achieve dove-tailing in a universal Turing machine).

Nevertheless, it is possible to derive an implementation automatically from a denotational semantic description. At least two prototype compiler writing systems based on denotational semantic description have been constructed [72, 52]. Automatic construction of code generator from a denotational semantic description is also the subject of much current research. Eventually it might be possible to construct production-quality code generators from a suitably restricted class of denotational semantics descriptions.

In an operational semantics, a language is defined by an interpreter. Often, the interpreter takes the form of an abstract machine, in which case the language semantics is specified as an algorithm for translating source program into abstract machine code. The *execution* of this code on the abstract machine traces out a sequence of states, and this sequence is taken to be the meaning of the program. We still have the problem of rigorously defining the abstract machine; that is, of giving a formal semantics for the abstract machine code. In a sense we have merely pushed the problem of semantic specification one level back.

Presumably the machine is so simple that there is no chance of misunderstanding its *semantics*. However, no standard formal basis exists for these methods – every new description might define a new machine or language on which to base the semantics. This makes it extremely difficult to compare programs since we cannot possibly run the abstract machines on all possible inputs exhaustively. Even worse, the meaning of a program is given only indirectly, by executing complete programs on an abstract machine or interpreter. The meaning of a fragment of a program, or of a particular syntactic construct in the language, is thus difficult to obtain.

Axiomatic frameworks are prone to incompleteness and inconsistency. The usefulness of axiomatic semantics is unfortunately severely limited by its inability to cope easily with common language features such as side effects and scoping. A more fundamental problem with this approach, at least as far as compiler or translatior generation is concerned, is that an axiomatic specification restricts one to reasoning about particular properties of programs – there is no stipulation that these properties correspond to program meaning [52]. It addresses only partial correctness with the assertions but is by no means a complete language specification. Hence, the

prospects for generating compilers or translators from axiomatic descriptions seem quite remote.

2.4.2 Algebraic semantics

In addition to the three formalisms, *algebraic semantics* [6, 27, 79, 55, 63] whose foundations are based on abstract algebra has received a good deal of attention. Algebraic semantics is a precursor of action semantics described in the next section. It involves the algebraic specification of data and language constructs. The basic idea is to name the different sorts of objects and the operations on the objects and to use algebraic axioms to describe their characteristic properties. Conceptually it is based on notions and ideas of classical and universal algebra in pure mathematics, and on concepts of abstract data types and software specification in computer science. The types in a programming language serve to classify the data processed by programs are called *sorts* in algebraic semantics. An algebraic specification defining one or more sorts contains two parts: the *signature* and the *equations* (or axioms). A signature Σ of an algebraic specification is a pair [*Sorts, Operations*] where *Sorts* is a set containing names of sorts and *Operations* is a family of function symbols indexed by the functionalities of the operations represented by the function symbols. The development of algebraic semantics is to consider classes of algebras satisfying some properties, such as the associativity of the statement sequencing operation.

Peter D. Mosses observed that denotational descriptions are, in some sense, "too concrete", due to the intertwining of model details with the actual semantics of the language [64]. He proposes a new algebraic approach to semantics in which

program meanings are given in terms of *abstract semantic algebra* (ASA's) [63]. The operators in an ASA belong to an algebraic sort of *actions*, which may produce and consume values, have side effects and so on. More importantly, the operators are chosen so as to directly reflect the fundamental concepts embodied by programming languages.

2.4.3 Action semantics

With a broad spectrum of definitional techniques available, action semantics is a formal method whose presentation is "gentle", providing just enough in the way of mathematical underpinnings to produce an understanding of programming languages. Action semantics is a form of denotational semantics that uses abstract semantic algebra (instead of Scott domains and λ-notation) to describe the values and denotations of programming languages. Mosses and Watt extended the idea of ASA's by developing a highly descriptive notation for expressing semantics of programming languages [65]. Their new method, *action semantics*, has been used to describe the semantics of Pascal [66] and ML [92, 58]. Their work is motivated by the observation that, despite the numerous advantages for relying on formal specifications of programming languages, readers generally avoid them when learning or implementing a programming language mainly because the formal notations are dense and cryptic. Most programmers understand programming languages in terms of basic concepts such as control flow, binding, storage update and parameter passing, but formal specifications often obscure these notions to the point that the reader must invest considerable time to determine, for example, how parameters are really passed or whether a language follows static or dynamic scoping. It is dis-

couraging that some of the most fundamental concepts of a programming language are the hardest to understand in a formal definition. The primary aim of action semantics is to allow useful semantic descriptions of realistic programming languages by using the familiar concepts in them; this is exactly what some semanticists have been advocating over the years: *making computer science concepts explicit in the formal description.*

Mosses [64, 79, 65, 66] suggested an attractive approach that seems to overcome most of the problems. The goal of his efforts has been to produce formal semantic specifications that are easy to read and understand, and directly reflect the ordinary computational concepts of programming languages. His idea is to identify a set of primitive and composite *actions* that correspond to fundamental concepts of computation in common programming languages. Examples of primitive actions are binding and finding values in the environment, storing and retrieving values. Examples of composite actions are composition, selection, iteration and abstraction of simpler actions. A given set of primitive actions, together with the operators that combine them into composite actions, is called by Mosses a *semantic algebra.* The actions have been chosen carefully both for their close correspondence to familiar semantic concepts and for their nice algebraic properties. The denotation of each syntactic phrase is an action, whose outcome describes what happens when the phrase is "evaluated".

The modularity of an action semantics description stems from defining the values and denotations of the programming language, and operations on these, to be elements of abstract semantic algebras. The action themselves form an abstract semantic algebra. This imposes a modular structure on the semantic description

that allows the semantic equations to be decoupled from the way the values and actions are defined. Consequently the semantic description is relatively easy to modify; if the described language is changed, and relatively easy to reuse for a related language.

Action semantics is a member of the functional semantics family characterized by the use of valuation functions that map the syntactic constructs of a language to their meanings which could be functions, numbers, algebraic terms, etc. The meaning of the entire program is determined by the valuation functions.

The primitive actions and combining operators of a semantic algebra can be defined axiomatically, in which case they constitute an *abstract semantic algebra*. The major advantage of this is that non-determinism (such as $a+b \rightarrow b$; $b+a \rightarrow b$ in an oriented equational reduction system) and concurrency ($a||b = b||a$) can be accommodated. The disadvantages are the usual problems of ensuring consistency and completeness of the axioms, and also the fact that the semantic description is not directly executable.

Alternatively, the primitive actions and combining operators can themselves be defined in λ-notation (or some dialect of it). Since there exist β-reduction machines for λ-terms, the semantic description is then executable, and much better structured than a conventional one. This is similar to our approach, described later.

Clearly, the choice of semantic algebra is crucial. We shall use a semantic algebra motivated by the algebra presented by Mosses. In general the semantic algebra will have to be tailored to the language being described, but it seems feasible to define a single semantic algebra that is adequate to describe a wide class of languages [52].

The action notation Mosses adopted has evolved from a more symbolic notation

to a more English-like presentation. The English-like notation of action semantics is somewhat easier on the reader, in our opinion, than the inference-rule notation of structural operational semantics, or the λ-calculs based notation of denotational semantics. Readability is a subjective issue, of course, being necessarily influenced by the reader's familiarity with the particular notation. However, we are convinced that a reader unfamiliar with action semantics can quickly reach the stage of understanding an action-semantics description, at least at a superficial level; and understanding of the semantic description will increase smoothly as familiarity with the notation is gained [64]. Although an action specification can be read at an informal level, it is indeed a formal definition. The main difference between action semantics and denotational semantics concerns the universe of semantic entities: action semantics uses entities called actions, rather than the higher-order functions in denotational semantics. Actions are inherently more operational than functions.

Action semantics framework is basically an algebraically oriented equational system. Extension, specialization, and abbreviations are all specified algebraically. There are three kinds of semantic entity used in action semantics: *actions*, *data* and *yielders*. Actions are essentially computational entities. The *performance* of an action represents dynamic information processing behaviors. Items of data are, in contrast, essentially static, mathematical entities, such as integers, boolean values, and abstract cells representing memory locations, that embody particles of information. Data are classified into sorts so that the kinds of information processed by actions are well specified in a language definition. Sorts of data are defined in the algebraic manner. A yielder represents an unevaluated item of data whose value depends on the current information, i.e., the previously-computed and input values

```
execute _ : Statement  →  Action [completing | diverging | storing]
            [using current bindings | current storage]

execute ⟦ if E then C1 else C2 ⟧ =
      evaluate E
   then
         check (the given TruthValue is true) and then execute C1
      or
         check (the given TruthValue is false) and then execute C2


elaborate _ : Declaration  →  Action [completing | binding | storing]
              [using current bindings | current storage]

elaborate ⟦ const I = E ⟧ =
      evaluate E
   then
      bind I to the given value


evaluate _ : Expression  →  Action [giving a Value]
             [using current bindings | current storage]

evaluate ⟦ I ⟧ =
         give the integer stored in the Cell bound to I
      or
         give the integer bound to I

evaluate ⟦ E1 + E2 ⟧ =
         evaluate E1
      and then
         evaluate E2
   then
      give sum (the given integer#1, the given integer#2)
```

Figure 2.1: Action semantics descriptions

that are available to the performance of the action in which the yielder occurs.

A few examples of action semantics description are shown in Figure 2.1. Action semantics notation uses indenting to describe the evaluation order of action operations. Parentheses are also allowed for this purpose, but indenting is generally preferred, being easier to follow. In Figure 2.1, `execute`, `elaborate`, and `evaluate` are

semantic functions whose arguments are statements, declarations, and expressions respectively. These examples illustrate a syntactic convention wherein parameters to operations are indicated by underscore. Operations in actions semantics can be prefix, infix, or outfix. Observe that one of the actions in the last semantic equation must fail, thereby producing either the constant binding or the variable binding to the identifier. The yielder

```
the given _#_ : Datum, PositiveInteger  →  Yielder
```

yields the n^{th} item in the tuple of transient data given to an action, provided it agrees with the sort specified as `Datum`, where n is the second argument. In Figure 2.1 the semantic equation `evaluate` only deals with integer type (or sort). To make it applicable to other types as well, the convention is to use the "value" sort. The sort `datum` in action notation is supposed to include all individual items of data used in a particular semantic description [64]. A common practice is to introduce the sort `value` ad hoc, corresponding to the informal concept of a value in a langauge. It is specified as

```
value = integer | real | character | string | □.
```

where □ indicates the sort `value` is left open for extension.

Action combinators, like `then` and `and then` in the above examples, are binary operations that combine existing actions, using infix notation, to control the order in which sub-actions are performed as well as the data flow to and from the subactions. Action combinators are used to defined sequential, selective, iterative, and block structuring control flow as well as to manage the flow of information between actions.

Another issue concerns the ubiquitous appearances of environments and stores in many semantic descriptions. In the formalism used in Milner's description of ML [58], the environments and stores occur explicitly in each evaluation rule. In some cases, the abbreviation that allows the store component to be elided from most of the rules does not entirely solve this problem; besides, the abbreviation is clearly *ad hoc*, and not generally applicable. In action semantics, by contrast, environments and stores never have to be specified explicitly; the action combinators take care of the various ways in which bindings and storage changes are sequenced or merged. These combinators are suitable for describing a wide variety of programming languages, not just ML with its left-to-right evaluation order for expression sequence.

Action notations can be manipulated algebraically using properties such as associativity, commutativity, and identity laws to prove the equivalence of certain action expressions. The action combinators, a notable feature of action notation, obey desirable algebraic laws that can be used for reasoning about semantic equivalence. And it is one of their strengths that they can provide a basis for sound reasoning about program correctness and equivalence.

Action notation uses the emphatic brackets "$\llbracket$" and "$\rrbracket$" slightly differently than denotational semantics in the sense that semantic functions are applied to abstract syntax trees. In action semantics the notation "$\llbracket E_1 + E_2 \rrbracket$" denotes the abstract syntax tree composed of a root and three subtrees, E_1, + and E_2. Since E_1 is already an abstract syntax tree, we have no need to wrap it inside another set of brackets when it is referred to in the right-hand side of the semantic equations.

Action notation is used to build a formal description of programming languages.

The semantic equations of the description define a translator from the source program into action notation, which can act as an intermediate language in an translator, compiler or interpreter. By providing an interpreter of action notation, we can obtain a prototype implementation of any programming language with a specification in action semantics. A translator of action notation into a machine language produces a compiler of the language.

Following denotational semantics, action semantics insists on compositionality with the semantic function mapping not only entire programs but also all their component phrases to semantic entities. It also conforms to a strict type discipline in specifying the meaning of language constructs. This careful delineation of the types of objects manipulated by actions adds to the information conveyed by the semantic descriptions.

The inherent modularity in an action semantics description smoothly scales up for describing practical languages. An action semantics description of one language can make widespread reuse of that of another, related language. Language definition modules will be highly interchangeable, and it will be possible to store the modules in a database for later use in other language designs. All these pragmatic features are highly appealing.

Equational reasoning is widely used in many applications including program transformation, partial evaluation, optimization, program reasoning, or programming development tools [34]. However, syntax and semantics of real programming languages do not lend themselves to "natural" equations. Mapping syntactic constructs of a language to its action semantics notation permits modular equational reasoning about programs.

2.4.4 High-level semantics

The style of semantic description used in our work is not exactly the same as that of action semantics. It is a variant of action semantics called *high-level semantics* originated by Peter Lee [52], in which *macrosemantics* refers to the semantic equations of the source language, and *microsemantics* the semantic algebra of the target language. Before showing some examples, we introduce some of the vocabulary of algebraic specification. The types in a programming language serving to classify the data processed by programs are referred to as *sorts*. An algebraic specification defining one or more sorts contains two parts: the *signature* and the *equations*. A signature Σ of an algebraic specification is a pair [*Sorts, Operations*] where *Sorts* is a set containing names of sorts and *Operations* is a mapping from function symbols to their profiles. The equations in a specification constrain the operations in such a way as to indicate the appropriate behavior for the operations. They serve as axioms specifying an algebra, similar to the properties of associativity and commutativity of operations that we associate with abstract algebra in mathematics. Equations may involve variables representing arbitrary values from various sorts in the specification. The variables in an equation are universally quantified implicitly. When applied to describe the semantics of a practical programming language, the model is similar to the conventional *many-sorted* (or *heterogeneous*) algebra.

There are three principle differences between high-level semantics and action semantics:

1. Operators in action semantics are directly defined by means of basic operators but in high-level semantics they are inductively defined in terms of basic operators. The advantages of having higher level operators in a semantic

description are similar to having subprograms in programming languages: a more modular and more succinct description. However, we should be careful when defining new operators so the users do not need to follow a long chain of definitions to find their definitions.

2. As in denotational semantics, compile-time and run-time aspects are not distinguished in action semantics. This makes the semantic specifications less descriptive and complicates the task of determining which portions of a semantics can be statically evaluated. In high-level semantics, the compile-time domains are clearly delineated from the run-time domains [52].

3. Instead of following the English-like description of action semantics, *outfix notation* is used in high-level semantics to make the presentation more terse and less confusing. The English-like description is too wordy; in high-level semantics appropriate operators are chosen to stand for commonly used action notation idioms. Because the action combinators in action semantics can be infix or prefix, readers may misunderstand the meaning of the description if they do not remember the precedence and parentheses are not present. Outfix notation gets rid of the problem completely.

High-level semantics is composed of two parts: macrosemantics and microsemantics. The major part of macrosemantics is a set of semantic equations mapping syntactic constructs to their meanings similar to the description of denotational semantics. An interpretation for the operators appearing in the right-hand side of the semantic equations is provided by the microsemantics part. Only the signature of the algebra defined by the microsemantics (i.e., the names and function profiles

```
action domains
   V_Action = _              (* value producing action *)

operators
   Integer : INT -> V_Action is
   Integer = _

   Add : V_Action * V_Action -> V_Action is
   Add = _

semantic functions
   E : expr -> ENV -> V_Action

semantic equations
   E [[ int ]] env = Integer (int)

   E [[ expr1 ``+'' expr2 ]] env =
      Add (E [[ expr1 ]] env, E [[ expr2 ]] env)
```

Figure 2.2: Macrosemantic description of a toy language

of the operators) is shared between macrosemantics and microsemantics. Semantic model details are used only in providing an interpretation for the algebra thus appear only in the microsemantics. This provides a clean separation of the actual language semantics from the underlying model details.

Consider the macrosemantics of expressions in a hypothetical toy language [52] shown in Figure 2.2: in the macrosemantics, the semantic equations map the language syntactic constructs to algebraic terms similar to those used in action notation, where an interpretation (or model) of the algebra will be given in the microsemantics. Only the microsemantics is dependent on the particular target language involved, so our translator G2A is easily retargeted to other languages. In the above example, the domain value-producing action, `V_Action`, along with operators `Add` and `Integer`, define a semantic algebra of value producing actions. Only

the signature of this algebra appears in the macrosemantics, but it provides enough information to allow us to write equations describing the semantics of integer expressions. A complete definition of the semantic algebra will be given separately in a microsemantic specification, thereby enforcing separability of the semantics from model-dependent details.

In traditional denotational semantics, the addition of a new language feature often requires a complete reformulation of the semantics. For example, in a traditional, direct-style semantics, the addition of escapes from loops causes the entire specification to be written. In a high-level semantics such extensions require only the addition of new equations to the macrosemantics [52]. All other changes are isolated in the microsemantics. Even though adding escapes from loops to a continuation-style denotation semantics requires revision of only a subset of the semantic equations, unfortunately it is often difficult to identify those equations needing modifications.

An interpretation for the operators is provided by a microsemantics. One possible (and very simple) microsemantic specification is shown in Figure 2.3, and a more general one [52] is given in Figure 2.4 wherein `intValue` and `rval` are value constructors for constructing and deconstructing values.

Shown in Figure 2.5 are two examples of semantic functions in a macrosemantics specification of a toy language [52], wherein function `D` is for the declaration of a variable of type integer, and function `E` is for the addition of two integer expressions.

The salient characteristics of our approach (some of which are mentioned by Lee and Mosses [52, 64]) can be summarized as follows:

Model independent language definition One problem with traditional deno-

```
action domains
   V_Action = STATE -> INT.

operators
   Integer : INT -> V_Action is
   Integer i = fn state . i

   Add : V_Action * V_Action -> V_Action is
   Add (v, v') = fn state . (v state) + (v' state)
```

Figure 2.3: A simple microsemantic description

```
action domains
   V_Action = STATE -> EV    (* EV stands for expressible values *)

operators
   Integer : INT -> V_Action is
   Integer i = fn state . rval (intValue i)

   Add : V_Action * V_Action -> V_Action is
   Add (va, va') =
      fn state .
         let
            rval (intValue v)  = va state
            rval (intValue v') = va' state
         in
            rval (intValue(v + v'))
         end
```

Figure 2.4: Another microsemantic description

tational semantics has to do with the lack of separation between the actual semantics of a language and the model-dependent details underlying it. Action semantics abstracts from the model-dependent details in order to concentrate on the real language features.

Accessibility A language description should be easy to understand, and to relate to familiar programming concepts, without a major effort in learning about

```
D [[ "var" id ":" "int" ]] env =
  if notDeclared(id,env) then
    let
      (* obtain a semantic name for the identifier *)
      varName = name(id).

      (* Construct the mode for the new variable *)
      mode = varM(varName, intType).

      (* Add the new declaration to the environment *)
      newEnv = addAssoc(id,mode,env).
    in
      (newEnv, DeclSimpleVar(varName,intType))
    end
  else
    declError [[id]] "Identifier already declared".


E [[ exp1 "+" exp2 ]] env =
  let
      (t1,v1) = E [[ exp1 ]] env
      (t2,v2) = E [[ exp2 ]] env
  in
    if (t1 = basicType(intType)) andalso (t1 = t2) then
      (basicType(intType), Add(v1,v2))
    else
      exprError [[...]] "can only add integers."
  end
```

Figure 2.5: Semantic functions in macrosemantic description

the description technique itself. It should not be necessary to be an expert in the idiosyncrasies of standard denotational semantics description techniques, for instance, continuations. Instead, one should be able to rely on knowledge of standard programming languages concepts in order to write and understand the semantics. Action semantics is suggestive of an operational understanding of the described language, and thereby making it easy to comprehend and possibly serving as a guideline for implementation.

Compared to other formalisms, such as the λ-notation, action notation may

appear to lack conciseness: each symbol generally consists of several letters, rather than a single sign. But the comparison should also consider that each action combinator usually corresponds to a complex pattern of applications and abstractions in λ-notation. For instance [64], the action term "A_1 *then* A_2" might correspond to something like the following CPS style λ-notation:

$$\lambda\epsilon_1.\lambda\rho.\lambda\kappa.A_1\epsilon_1\rho(\lambda\epsilon_2.A_2\epsilon_2\rho\kappa)$$

Action combinators are binary operators that combine existing actions, using infix notation, to control the order in which subactions are performed as well as the data flow to and from the subactions. The action combinator *then* is an action combinator that performs the first action; when it completes, the second action is performed, taking the data given by the first action. In any case, the increased length of each symbol seems to be far outweighed by its increased readability and expressiveness. The readability of an action semantics description stems primarily from the close links between the action notation and familiar semantic concepts, and secondarily from the use of an English-like notation for the actions.

Ease of language update If part of the source language definition is changed at a future time, instead of working on the ad hoc translation program, we can simply change the macrosemantics to reflect the change (this is analogous to the current situation with regard to syntax; given a context-free grammar for a language, we can apply tools to automatically generate a parser for that language). The description is easily modifiable, in the sense that language

changes (such as removal or addition of imperative features, or changes in evaluation order) would require only commensurate changes to the description — rather than forcing it to be completely rewritten.

Exposition of implementation structures The operators of our semantic algebra are chosen to directly reflect both fundamental language concepts as well as fundamental implementation concepts. This improves the comprehensibility of the semantic descriptions. Furthermore, an efficient implementation can be obtained by interpreting the operators as templates of intermediate code for a code generator.

Separability The semantic equations and the semantic algebra are defined in separate specifications called the macrosemantics and microsemantics, respectively. Semantic model details are used only in providing an interpretation for the algebra, and thus appear only in the microsemantics. This provides a clean separation of the actual language semantics from the underlying model details.

Interchangeability of microsemantics definitions A nice feature of high-level semantics is its interchangeability. We can plug in different microsemantic specification for the same macrosemantics. The only information shared between the macrosemantics and microsemantics is the signature of the semantic algebra defined by the microsemantics. This provides a modularity that guarantees the invariance of the macrosemantics under different interpretations of the microsemantics operators.

It is straightforward to re-target a translator to other languages when high-

level semantics is used as translational semantics: a separate microsemantic description suffices.

Distinction between static and dynamic language components The separation between macrosemantics and microsemantics is also used to distinguish between the static and dynamic aspects of a language. Static semantics treats context-sensitive syntax (eg., declaration of identifiers before use or well-typedness of expressions) as a kind of semantics. It is called static semantics because it depends only on the program structure and but not the program input. Dynamic semantics refers to the input-dependent behavior of a program and it may be defined independently of the static semantics of the program. G2A allows user-parameterized specification of primitive types and operators in the microsemantic specification. In this respect, high-level semantics is similar to the two-level semantics proposed by Nielson and Nielson [67].

Extensibility It is usually straightforward to add new operators to a semantic algebra. Doing so may require rewriting of parts of the microsemantics, but always leaves the macrosemantics intact. The portions of the microsemantics requiring modification are easy to identify.

Executability Similar to what David Watt achieved in [91], the semantic description in the macrosemantics is actually executable, i.e., yields an executable translator. This makes it very different from other frameworks like *Formalism* **Z** [82] which is based on non-constructive typed set theory.

Facilitation of mathematical reasoning Formal descriptions can be used as

the basis for systematic development and automatic generation of implementations. It is one of their strengths that they can provide a basis for sound reasoning about program correctness and equivalence.

Expressiveness Action notation is designed with sufficient primitives and combinators for straightforwardly expressing most common patterns of information processing in programming languages, i.e., it is not necessary to simulate one kind of information processing by another.

Reusability For economy of effort, language designers want to be able to reuse parts of descriptions of existing languages in the description of a new language. In our approach, language definition modules are highly interchangeable, and it is possible to store the modules in a database for later use for other language designs.

Avoidance of over-specification A problem arises in defining the *collateral* action $\mathcal{A}_1$ & $\mathcal{A}_2$. By definition of the term "collateral", no particular order should be defined for performing $\mathcal{A}_1$ and $\mathcal{A}_2$. In denotational or operational semantics, however, there is no simple way to avoid over-defining the sequence of actions. The reason is that both denotational or operational semantics are inherently sequential [8]. The semantic algebra approach has the advantage that this particular problem is localized and isolated, and does not intrude upon the top-level semantic description of the programming language. Certain details can be deliberately left unspecified in high-level semantics description such as the evaluation order of subexpressions or function parameters to make the specification more flexible.

Straightforward re-targeting To re-target to other languages, a separate microsemantic description suffices.

Chapter 3

Semantics-based language translation

Program translation requires mapping the semantics of the source language to that of the target language. Semantics-based language translators provide formal semantic descriptions for validating prototype implementations of the source languages under design. This is of paramount importance for a language which is still being designed and implemented. The semantics based approach helps to ensure that the final implementation does, in fact, adhere to the specification in a controlled fashion.

A problem that is seldom addressed in the development of prototyping languages is teamwork. One of our goals in the development of language translators is for easing teamwork on the dynamically evolving source language.

Many language translators lack structure and are hacked unmercifully, thereby making them difficult to understand, let alone to maintain. It is desirable for the

structure of the translator to support both modifiability and extensibility, which in turn permits a quick development of language translators.

3.1 Motivation and issues

Every software developer knows that it is almost impossible to keep an evolving software system consistent with the corresponding documentation. For this reason documentation is frequently written at the end of the implementation activity. Most of the time it is not worth writing extensive documentation in the course of the development of a prototyping language venture.

As most programming languages become more expressive and the list of features they provide continues expanding, it is more and more difficult for language implementors to stay abreast of the exact meaning of language constructs, and increasingly more important that the semantics of the language be communicated unambiguously.

In giving formal semantics to a programming language we provide a basis for understanding and reasoning how programs behave. Not only is a mathematical model useful for various kinds of analysis and verification, but also, at a more fundamental level, because merely the activity of trying to precisely define the meaning of program construction highlights the inconsistency of the language design in early stages. Carefully chosen, the semantic description often has the advantage of abstracting away from unimportant details, as well as providing higher-level concepts with which to understand the dynamic computational behaviour of a program.

3.2 Formal semantics based language translators

By semantics based language translators we mean those translators driven by semantics [40, 12, 46, 53, 88]. Usually they are structured as a series of relatively simple transformations, each of which is semantics-preserving within some formal semantics framework.

3.3 Action semantics in action: structuring a language translator in action semantics framework

Denotational semantics has poor pragmatic features, due to its use of the lambda-notation [52]. In action semantics, the meaning of a programming language is defined by mapping program phrases to actions, which directly reflect the ordinary computational effects of programming languages that are easy to read and understand. The performance of these actions models the execution of the program phrase.

The steps involved in our translation can be roughly delineated as:

- Analyze the source program according to the static semantics description defined in the macrosemantics of the source language.
- Obtain the intermediate representation in a dialect of action notation after the static analysis.
- Generate the target code according to the intermediate representation.

Shown in Figure 3.1 and Figure 3.2 is an example briefly outlines the translation of a function application. The function `sum` is of type `int * int -> int`. The

semantic equation `E` (of type `expAST -> (semType * vA)`) maps an expression AST to a tuple, whose first element is the static semantic type of the expression and second element the action denoting the entire expression. Sequence of expressions are handled by `Es`, which is of type `expAST list -> (semType * vA) list`.

In the rest of this section we illustrate the derivation of the action representing the function call `sum(1,2)` according to the semantic equations shown in Figure 3.1. The format of macrosemantic equations is similar to that of the traditional denotational semantics. We use the emphatic brackets ([[]], or textually "`[[ ]]`") around an argument of a semantic function to show that the argument is a syntactic phrase.

The valuation of the function application `sum(1,2)` begins with the semantic equation for function application, in which type inference (checking) is first performed. Functions in Griffin always takes one argument so the type of the argument of function `sum` is of tuple type `int*int`.

The intermediate representation, TPOT (described in more detail in Section 4.1), is obtained according to the semantic valuation functions. Figure 3.2 gives the syntax definition for actual parameters and the type of the action combinator `and`. Eventually we arrive at the action corresponding to the Griffin function call `sum(1,2)`.

Since Griffin does not specify the evaluation order of subexpressions or function parameters, the action combinator `and` is used rather than the order-establishing combinator `and then`. The action combinator `then` (syntax: A_1 `then` A_2) performs the first action using the transients given to the combined action and then performs the second action using the transients given by the first action. The transients given by the combined action are those given by the second action. For action

Griffin code

```
fun sum (x:int, y:int) => x+y;
sum(1,2);
```

related macrosemantic equations

```
(integer)
    E  [[  int  ]]  env  =  (intTy, Integer int)

(function application)
    E  [[  expr  expr'  ]]  env  =
       let
          (FunType(domainType,rangeType), vA) = E [[ expr ]] env
          (t', vA') = E [[ expr' ]] env
       in
          if unify(domainType,t') then (rangeType, FuncAppl(vA,vA'))
          else error("function application type mismatch)
       end

(parenthesized expression sequence)
    E  [[  "("  expr  ","  exprs  ")"  ]]  env  =
       let
          (t, vA)  = E  [[ expr ]]  env
          (ts,vAs) = Es [[ exprs ]] env
       in
          (TupleType(t::ts), ExprSeq(vA,vAs))
       end

(expression sequence)
    Es  [[  expr  ","   exprs  ]]  env  =
       let
          (t, vA)  = E  [[ expr ]]  env
          (ts,vAs) = Es [[ exprs ]] env
       in
          (t::ts, ExprSeq(vA,vAs))
       end

    Es  [[  ]]  env = (nil,NullExpr)
```

untyped TPOT for the function call

```
FuncAppl(Id sum, ExprSeq(Integer 1, ExprSeq(Integer 2, NullExpr)))
```

Figure 3.1: From Griffin to TPOT

action semantics description

```
Association  =  [[  [identifier "=>"]  Expression  ]].

_ and _  ::  action, action   →   action
                (total, associative, unit is complete)

application of _ to _  ::  yielder, yielder   →   action

enact  ::  yielder    →   action

evaluate _  ::  Actuals  →  action
   [giving arguments | diverging | storing]
   [using current bindings | current storage].

evaluate <E: Expressions "," A:Associations> =
   evaluate E and evaluate A.
```

action representing the function call

```
      give 1
   and
      give 2
then
   enact application of (the Function bound to sum) to the given value
```

Figure 3.2: Action semantics in action

combinators `and then` and `then`, if one of the actions gives the value `nothing` (undefined, bottom), the result of the composite action is `nothing`, that is, these combinators are strict in the value `nothing`. The execution semantics of Griffin is strict, thus the combinator `then` is chosen to reflect that.

The action A_1 `and` A_2 represents implementation-dependent order of performance of the indivisible subactions of A_1 and A_2. When these subactions cannot "interfere" with each other, it indicates that their order of performance is simply

irrelevant.

In action semantics an *abstraction* is a datum that merely incorporates a particular action. It corresponds to the "code" for the action, which could be implemented as a sequence of machine instructions, or as a pointer to such a sequence [64]. Abstractions represent the semantics of programming constructs such as functions or procedures.

The yielder `application of` Y_1 `to` Y_2 affixes the argument value yielded by Y_2 as the transient that will be given to the action encapsulated in the abstraction yielded by Y_1.

The action `enact Y` performs the action encapsulated in the abstraction yielded by `Y`, using the transients and bindings that are included in the abstraction.

The most important advantage of this approach results from the action notation intermediate representation. All the syntactic constructs are defined in the action semantics framework, the denotations of them are absolutely formal and unambiguous. This is crucial for a language which is still being designed and implemented; as a clear documentation of language semantics prevents the misunderstanding between language designers and implementors. Inconsistent specification may give rise to false conclusions in reasoning, thus destroying the point of having a specification.

As a consequence of the way our translator is organized, the action semantics description of the source language provides a formal but more importantly, a high-level and intuitive description of the language semantics which contributes to the readability and maintainability of the language. The formal semantics notation can also be used as an intermediate representation when writing a translator, interpreter, or compiler. Furthermore, the description itself is directly executable so

it can be used to automatically obtain a static semantics analyzer.

The provision of a structural operational semantics for action notation emphasizes the operational essence of action notation, and allows the verification of algebraic laws.

3.4 Type-preserving translation

It is desirable that the target language of a translation is typed, and the translation satisfies the constraint that any well-typed source program will be mapped into a well-typed target program. Similar to translations based on typed intermediate languages, translations into a typed target language have certain software-engineering advantages: types help reduce static errors in the target code, and thus shorten the development cycles of a translator (especially in the debugging and maintenance stages). This is one prominent reason that typed intermediate languages is gaining increasing popularity.

A *type-preserving translation* maps expressions of the same source type to expressions of the same target type. The target code of a type-preserving translation is *typable* [88].

A type-preserving translation makes it easier to write the translation in a compositional fashion which is typed. A translation is compositional if the translation of a compound construct can always be specified in terms of the translations of its immediate components, i.e., the translations form a congruence (Section 4.5):

$$[\![f(t_1, \ldots, t_n)]\!] = [\![f]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$$

for all syntactic constructions f and compound constructs $f(t_1, \ldots, t_n)$ built from

component constructs t_1 to t_n. If the translation is type-preserving, then the type of $[\![f]\!]$ is solely determined by the types of source terms t_i, independent of the values of these terms.

Like other compositional specifications, a compositional translation is more amenable to design, understanding and reasoning. Compositionality helps break down the dependencies of compound terms and their subterms, so that each construction can be specified and understood without intensional reference to specific component terms. On the other hand, compositional translations simplifies the reasoning about the properties of the translation through *structural induction* over the syntax. This induction principle reduces the proof of a property that holds for all components in an inductively defined set (the syntactic category in our case) to the proof of this property for all compound constructs assuming the property holds for their components.

One further advantage of the type-preserving translation is its support for separate compilation. In general, the signature (or interface, contract) of a program module A gives only the type information, not the program body. In a setting that does not preserve types, the type of the target code for module A is not fixed; consequently, any other module B that uses module A cannot assign a unique type for references to entities in the interface part of A. It is possible to pass the translated types around at run-time, but it incurs high run-time overhead. In contrast, in a type-preserving setting, the target code of module A has fixed types; any other module B uses A can have uniquely typed references to entities declared in the interface, and types do not need to be passed at run-time, making separate compilation more practical. By the same reasoning, the translation need not insist on

whole-program transformation.

The following example illustrates a type-preserving translation. Consider the translation of the Griffin program shown in Figure 3.3(a), wherein `round` is the rounding operation mapping reals to integers. Both `x` and `y` have the same *existential type* (Section 5.1.3 and Section 7.2) in the source program. Existential types are typeset in the form $\exists\alpha.\tau$, as they appear in standard literature, rather than actual Griffin syntax.

```
fun truthValue true  => 1
 |  truthvalue false => 0;

var x : ∃ α . α × (α → int)
   = pack α=bool in  α × (α → int) (true,truthValus) end pack;

var y : ∃ α . α × (α → int)
   = pack α=real in α × (α → int) (3.45,round) end pack;

fun foo p => unpack p as (v,f) in f v end unpack;

foo x;   ---  evaluates to 1
foo y;   ---  evaluates to 3
```

(a) Griffin program

```
fun truthValue true  = 1
 |  truthValue false = 0

val x = (true,truthValue)
val y = (3.45,round)

fun foo (v,f) = f v;   (*  universal type  *)

foo x;   (*  evaluates to 1  *)
foo y;   (*  evaluates to 3  *)
```

(b) ML-like target program

Figure 3.3: An non-type-preserving translation

For the sample Griffin program shown, a non-type-preserving translation based on naive universal type encoding [40] is sufficient. However, it will not work in the following Griffin program,

```
var x : ∃α.τ := e1;
var y : ∃α.τ := e2;
...
x := y;
```

a compile-time type error may occur in a non-type-preserving translation because `x` and `y` may be of different types in the target program. Nevertheless, the target code can still be typable if run-time tags are attached. But again, this incurs run-time overhead, thus impractical.

Chapter 4

The G2A translator

G2A translates Griffin programs into Ada programs. Similar to other compilers and translators, it can be characterized as a process of analysis followed by synthesis. These two steps are further divided into several phases, described in the next section, which can be viewed as the structural components of G2A.

When it comes to the maintenance problem of the translator, the way we structure the language translator is more modular because it is based on the action semantics which is characterized by its modularity. Action semantic description are divided into *modules*, which, in larger specifications, may themselves be divided into *submodules*, just as we normally divide technical reports and textbooks into sections and subsections. It is often helpful to divide the modules of semantic description into submodules. For instance, suppose that we are describing the semantics of Pascal. We might divide the abstract syntax module into submodules concerned with expressions, statements, and declarations. Similarly for the corresponding semantic function modules. We could also divide the semantic entities module into submodules dealing separately with numbers, arrays, procedures, etc.

Such submodules might be reusable, with minor modification, in other semantic descriptions. Consequently, it is easier to modify the translator the way we organize it.

4.1 Overview of G2A

The first phase of G2A is the *syntax analysis* in which the front end, **Parser**, has the following functionality

$$\textbf{Parser} \quad : \quad \mathcal{P}_{\text{Griffin}} \rightarrow \mathcal{AST}_{\text{Griffin}}$$

where $\mathcal{P}_{\text{Griffin}}$ is the domain of textual representation of Griffin programs, and $\mathcal{AST}_{\text{Griffin}}$ is the domain of Griffin abstract syntax tree representations. Syntax analysis is further split into three phases: lexical analysis, parsing, and tree building. **Parser** examines the syntactic structure of a Griffin program, rejects it if incorrect, otherwise builds an abstract syntax tree representation of it. The source program is first parsed by the parser generated by the front-end generator (FEG) SML-LEX and SML-YACC according to the lexical and syntactic specification of the source language Griffin.

We have to define the macrosemantics of Griffin so that Griffin programs can be analyzed in the action semantics based framework. The semantic description of Griffin covers only the bare language directly; the remainder of the language is described by syntactic transformation down to the bare language. The description style of macrosemantics is similar to that of denotational semantics, which is mapping syntactic constructs to their "denotations". The denotation in the Griffin macrosemantics specification corresponds to an intermediate representation ($\mathcal{IR}$)

between the front end (Griffin programs) and the back end (Ada programs). This intermediate representation is in the form of *typed prefix operator terms* (TPOTs). There has been much recent interest in using typed intermediate representations in compilers, but in most cases types are abandoned well before code generation. G2A does keep type information in TPOT in the translation process; one reason is that it is necessary for the generation of temporary variables since Ada is explicitly typed and not an expression language. A TPOT represents the model-independent meaning of a program component. For example, the TPOT that represents an assignment expression $x := 1$ would look like

Assign(Var(x,IntType), Integer 1) : void

The type void at the end of the TPOT refers to the type of the assignment expression. These type annotations suffice to reconstruct the types of arbitrary terms.

The second phase is the *static semantic analysis and intermediate code generation* performed by the *translator kernel*, **TK**, which converts Griffin ASTs to TPOTs.

$$\mathbf{TK} \quad : \quad \mathcal{AST}_{\text{Griffin}} \rightarrow \mathcal{IR}_{\text{TPOT}}$$

Semantic model details still need to be supplied in order to define the meanings of the operators in a microsemantics, but this can be done without disturbing the macrosemantics developed.

We call the semantic metalanguage used in the Griffin macrosemantics specification *TML*, a tiny subset of *ML* [58, 73] with some minor syntactic changes. ML originated as the metalanguage of Edinburgh LCF, but has evolved into a programming language in its own right. ML is basically a functional language,

with functions as first-class objects, thus allowing higher-order functions and partial application (currying). It has a static type system, declared objects need not be explicitly typed, but their types are deduced by the compiler if necessary. ML provides powerful and high-level control mechanism and algebraic (symbolic) data types. It is unreasonable to expect readers of the Griffin macrosemantics specification to learn the complete ML language in order to comprehend the semantic equations, thus we have kept TML as small and applicative as possible to keep its complexity substantially lower than that of ML.

TK analyzes abstract syntax trees according to the syntax-directed equational specifications in the macrosemantics in order to collect the statically determined information about a program, then generates the TPOT, which is similar to an action notation term except that it is decorated with type information. The TPOTs contain operators that are defined in the back-end microsemantics and are suitable for target code generation. The code for Griffin's type checker used in the macrosemantic specification was written by Edward Osinski [70]. The semantics given in the macrosemantic specification defines the behaviour of well-typed programs only (types play no part in the dynamic semantics of such programs).

Finally, the Ada code generator is as follows:

$$\mathbf{CG} \quad : \quad \mathcal{IR}_{\mathrm{TPOT}} \rightarrow (I \rightarrow O)_{P_{Ada}}$$

where $(I \rightarrow O)_{P_{Ada}}$ is the universe of representations of Ada programs from the input domain I to the output domain O. To be more specific, the intermediate representation, TPOT, is transformed into an $AdaAST^{+}$, which is an abstract syntax tree representation of an *extended Ada program*. By extended Ada programs we mean programs similar to Ada programs but do not conform completely to the

Ada syntax definition. A *term rewriting system* [25] converts the $AdaAST^{+}$ into an Ada abstract syntax tree, $AdaAST$, that conforms to the syntactic rules of Ada. More details about the term rewriting system and code generation can be found in Section 7.14. In the final stage, *a multi-level pretty printer* [45] turns $AdaAST$ into a textual representation of a formatted Ada program.

The functional composition of these three phases yields **G2A**, a translator from Griffin to Ada:

$$\mathbf{G2A} = \mathbf{Parser} \circ \mathbf{TK} \circ \mathbf{CG} \quad : \quad \mathcal{P}_{\text{Griffin}} \rightarrow (I \rightarrow O)_{P_{Ada}}$$

G2A does not directly support multiple Griffin components in a program, although functions generated from one Griffin component can be treated like any other Ada functions and imported into another Ada component. A significant restriction of G2A is that it requires access to the entire Griffin program, for reasons summarized in Section 8.1. However, we believe that this problem can be at least partly addressed by providing separately compiled components a digest of the relevant type and function information from other components.

As in most formal semantics frameworks, abstract syntax is used in the specification since it represents the compositional structure of phrases of programs, ignoring how that structure might have been determined. In general, it is easier to define the semantics of programs on the basis of their abstract syntax, rather than on their concrete syntax.

A macrosemantic specification, along with a microsemantics interface file, is processed by the semantic analyzer to generate the **TK**. The abstract syntax tree (AST) specification is also fed to the semantics analyzer so **TK** can understand the AST output from the front-end.

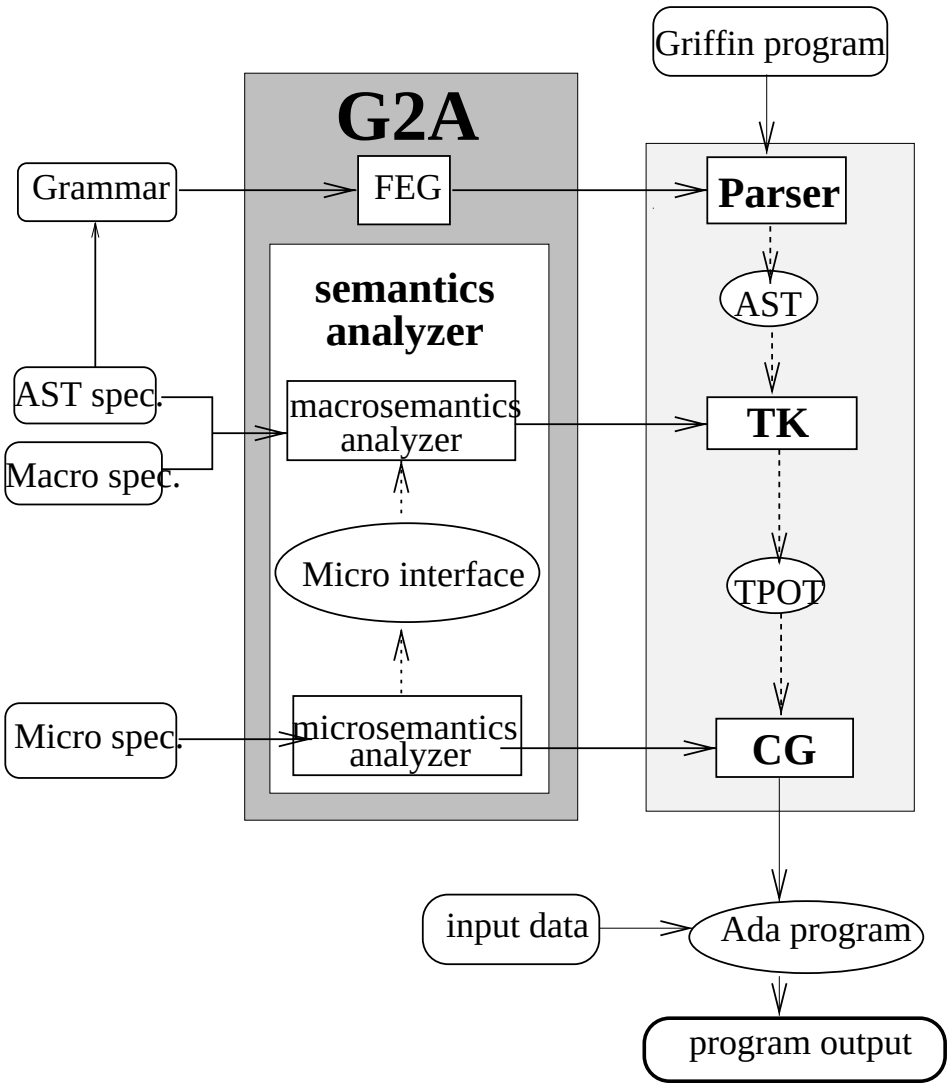


Figure 4.1: The schematic representation of the system shows the various phases of the translation process.

Both the macrosemantics and microsemantics analyzer are written in ML, using the NJ/SML implementation, so is the automatically generated translator kernel. **TK** translates abstract syntax trees into typed prefix operator terms (TPOTs). The code generator (CG) implements the semantic algebra; it plays the role of

emitting Ada code from the TPOT in G2A.

A schematic representation of G2A (adapted from Peter Lee's MESS system diagram [52]) is shown in Figure 4.1.

It is worth pointing out that although using our approach may seem natural, it is not the only possibility. Suppose there are n languages and we would like to translate one to another. Another possible approach is to write a translator from each language to the other $n-1$ languages. With n languages in the picture, $n*(n-1)$ different translators must be written, instead of just $2n$ for going to and from TPOTs as in the case of G2A.

4.2 Front-end reuse

An important factor attributable to the success of the popular compiler `gcc` is its well-engineered idea of the decoupling of its front-end from its back-end. Reusing general infrastructure components is probably the easiest way and low-risk way to develop front-ends.

In the design phase of the abstract syntax tree, pragmatically reusable components should be identified and incorporated into the current design. If the abstract syntax tree definition has rich syntactic constructs, it may well be used for other source languages. This gives translator writers for other source languages greater flexibility to actually reuse and adapt parts of existing components. Thus it increases the productivity of translation program writers.

If the collection of abstract syntax trees are designed to be general enough, it can be used for other languages as well. For example, the following fragments declare a variable "x" of integer in various languages:

```
var x : int;        --  in Griffin
x : integer;        --  in Ada
int x;              /*  in C, C++  */
x : integer;        /*  in Java  */
```

In some languages, an initializing value can be specified in variable declarations. In our implementation, the abstract syntax tree for variable declaration is

```
declAST  ::  VarDecAST of id:string, type: typexpr, initExpr: expr
```

which covers all the cases above. Similarly, there are unimportant syntactic differences in looping constructs (while loops, for loops, or unconditional loops) and other syntactic constructs. As long as the abstract syntax trees are designed to accommodate the most involved one, they can be used for various source languages. Naturally the very same abstract syntax trees definition of a language can be used for the back-end code generation process (UN-parsing) as well.

However, a requirement to reuse the components in our systems is that ML has to be the choice of the implementation language because all the building blocks of our system are written in ML.

4.3 Generic backend

The implementation of the translator should also permit efficient execution of programs written in the source language. This can be achieved in G2A by choosing the appropriate microsemantic specification, which furnishes the semantic algebra for "operators" used in the TPOTs. We can also perform conventional optimization phases to the interpretation of TPOT in order to improve run-time efficiency.

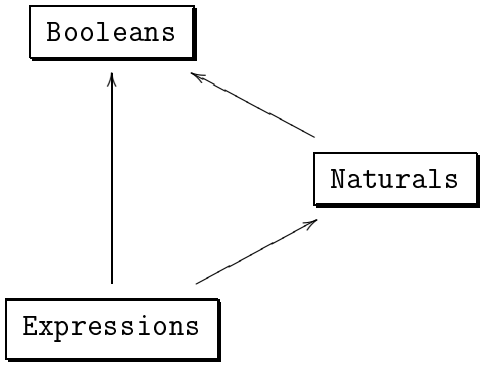
A side product of the generic backend is the possibility of *semantic recovery*: other formal descriptions (e.g., λ-calculus terms, operational semantics) of the source language can be derived from action notation [52].

4.4 Semantic modules reuse

Action semantic descriptions are divided into modules. Larger specifications may be divided into submodules. A common practice is each module has to be self-contained.

In action semantics the semantic modules are expressed in algebraic specification involving operations and sorts. The basic principle involves describing the logical properties of data objects in terms of properties of operations that manipulate the data. The actual representation of the data objects and the implementations of the operations on the data are not part of the specification.

Shown in Figure 4.2 are three algebraic specifications defining truth values, natural numbers and expressions. The dependency among them is as follows:



```
module Booleans
   exports
      sorts Boolean

      operations
         true          : Boolean
         false         : Boolean
         not _         : Boolean -> Boolean
         and _,_       : Boolean,Boolean -> Boolean
         or _,_        : Boolean,Boolean -> Boolean
         implies _,_   : Boolean,Boolean -> Boolean
         xor _,_       : Boolean,Boolean -> Boolean
         eq _,_        : Boolean,Boolean -> Boolean
   end exports

   variables
      b,b1,b2 : Boolean

   equations
      and(true,b)       = b
      and(false,true)   = false
      and(false,false)  = false
      not(true)         = false
      not(false)        = true
      or(b1,b2)         = not(and(not(b1),not(b2)))
      implies(b1,b2)    = or(not(b1),b2)
      xor(b1,b2)        = and(or(b1,b2),not(and(b1,b2)))
      eq(b1,b2)         = not(xor(b1,b2))
end Booleans
```

```
module Naturals
   imports Booleans

   exports
      sorts Natural

      operations
         0             :  Natural
         succ _        :  Natural -> Natural
         lessthan _,_  :  Natural,Natural -> Boolean
         eq _,_        :  Natural,Natural -> Boolean
         ...
end Naturals
```

```
module Expressions
   imports Booleans Naturals
   exports
      sorts Expressions
      ...
end Expressions
```

Figure 4.2: Algebraic specification modules

The specification of `Naturals` (natural numbers) relies on that of `Booleans` (truth values). `Expressions` needs to access both `Naturals` and `Booleans`. Conceivably, the hierarchy continues as `Expressions` specification can be imported into `Statements` and eventually the `Programs` specification module for the entire program. Conventionally, the dependency among algebraic specifications forms a directed acyclic graph (DAG). If the dependency relation contains a loop, we have an undesirable meta-circularity.

For economy of effort, language designers want to be able to reuse parts of descriptions of existing languages in the description of a new language. The inherent modularity in an action semantics description smoothly scales up in describing practical languages. An action semantics description of one language can make widespread reuse of that of another, related language. It is possible to store the modules in a database for later use in other designs. We could discuss several further points of contrast between action semantics descriptions, for instance the extent to which descriptions can be recycled, so as to reduce the amount of new material needed when describing new languages. Obvious candidates are algebraic specifications for standard sorts of data (eg., boolean, integer, real, character, string).

4.5 Program equivalence and correctness concerns

First we briefly review the notion of *congruence* ($\cong$) for the discussion in this section. The constants of the universe in an algebra create a set of ground terms, and the equations of an algebraic specification generate a congruence $\cong$ on the ground terms. A congruence is a stronger equivalence relation with an additional "substitution" property. Let $\mathcal{S} = < \Sigma, \mathcal{E} >$ be an algebraic specification with signature Σ and

equations $\mathcal{E}$. The congruence $\cong_{\mathcal{E}}$ determined by $\mathcal{E}$ on the term algebra $\mathtt{T}_\Sigma$ is the set-theoretically smallest relation (with regard to set-containment partial ordering) satisfying the following properties:

Variable assignment The equations of $\mathcal{E}$ are of the form $\mathtt{lhs} \overset{\mathcal{C}}{=} \mathtt{rhs}$. For the equation to be true, the condition $\mathcal{C}$ must be satisfied ($\mathcal{C}$ is often tautology). Given an equation $\mathtt{lhs} \overset{\mathcal{C}}{=} \mathtt{rhs}$ in $\mathcal{E}$ that contains variables $v_1, \ldots, v_n$ and given any ground terms $t_1, \ldots, t_n$ from $\mathtt{T}_\Sigma$ of the same sorts as the respective variables,

$$\mathtt{lhs}\,[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n] \;\cong_{\mathcal{E}}\; \mathtt{rhs}\,[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$$

where $v_i \mapsto t_i$ indicates substituting the ground term t_i for the variable v_i. If the equation is conditional, the condition must be valid after the variable assignment is carried out on the condition.

Reflexivity For every ground term $t \in \mathtt{T}_\Sigma$, $t \cong_{\mathcal{E}} t$.

Symmetry For any ground terms t_1, $t_2 \in \mathtt{T}_\Sigma$, $t_1 \cong_{\mathcal{E}} t_2$ implies $t_2 \cong_{\mathcal{E}} t_1$.

Transitivity For any terms t_1, t_2, $t_3 \in \mathtt{T}_\Sigma$, $t_1 \cong_{\mathcal{E}} t_2$ and $t_2 \cong_{\mathcal{E}} t_3$ implies $t_1 \cong_{\mathcal{E}} t_3$.

Substitution property Suppose $t_1 \cong_{\mathcal{E}} t_1', \ldots, t_n \cong_{\mathcal{E}} t_n'$, $S, S_1, \ldots, S_n$ are sorts, for $i \in 1 \ldots n$, $t_i, t_i' \in S_i$, and $f : S_1, \ldots, S_n \to S$ is any function symbol in Σ. Then $f(t_1, \ldots, t_n) \cong_{\mathcal{E}} f(t_1', \ldots, t_n')$.

The correctness of our translation process would have to be established by a tedious congruence proof involving the microsemantics which defines the operators. To do this, the code generator specification needs to be broken up into several

passes, each of which implements a small set of transformations on the TPOT. On the whole, however, the indicated separation of the code generator specification into several modules would vastly simplify a proof for the entire translator. The number of transformation steps, i.e., the proof length, is very large (that may be one reason why no one has succeeded in providing a correctness proof for `gcc`) [52].

An important feature of action semantics is that it facilitates the mathematical reasoning of program semantics in the algebraic framework. The operational semantics of action notation determines the processing of each action. But this does not, by itself, provide a useful notion of equivalence between actions. From a user's point of view, however, two actions may be considered equivalent whenever there is no *test* (or *observation of interest [93]*) that reveals the differences in their processing.

We expect that testing equivalence of actions to include various algebraic laws, such as associativity of the action combinators. Moreover, we expect it to be a congruence, i.e., preserved by the combinators. The given algebraic laws facilitate the algebraic reasoning to show that various compound actions are equivalent, perhaps justifying a *program transformation rule* for some language on the basis of its action semantics.

Chapter 5

Overview of Griffin

In this chapter a brief introduction to the source language, Griffin, that we experiment with, is presented. In Section 5.2, we outline the building of our intermediate representation, TPOT, according to the macrosemantic specification of Griffin.

5.1 The source language: Griffin

Prototyping languages are programming languages appropriate for constructing executable prototypes for development-intensive software. The most important property of a prototyping language is its ability to permit the exploration of the problem and solution space at low cost in early software development stage, which means it should allow the expression of the essential content of an algorithm while leaving out the unnecessary details. Griffin is a broad-spectrum and statically typed prototyping language with strict execution semantics, intended for prototyping software that will eventually be written in Ada (most likely Ada95), featuring strongly typedness, parametric overloading, concurrency, exception handling, data encapsu-

lation, object-oriented paradigm [75], modules for *programming in the large*, pattern matching and expressiveness. We view Griffin as a descendent of SETL [77], Ada [9], C++ [28, 85], ML [58], Haskell [43] and an informal introduction to it is given in the following sections.

5.1.1 Basic language syntax

Griffin is a block structured language and its control flow constructs can be found in most imperative languages. Figure 5.1 gives a simplified version of Griffin's abstract syntax. In this and other syntax descriptions, we use the notation $\langle x \rangle_{sep}$ to mean a sequence of zero or more x's separated by *sep*, $x \mid y$ for alternatives x or y, and $[x]$ to mean an optional x.

Case expression is multiple-branch conditional which provides pattern matching on the argument expression (discussed in more detail in Section 5.1.9). Introduction to the iterators and generators used in the loop expressions also appears later (Section 5.1.7 and 7.12).

The expressions appearing in a function body or a let expression are evaluated sequentially, and the result of the last expression in the body is returned as the result of the expression sequence.

5.1.2 First-class functions

Griffin supports first-class functions which enables programming at a higher level of abstraction. In Griffin, procedures are simply functions whose return type is *void*. Pattern matching with user-defined deconstructors can be used in function definitions. The formal parameter types and the return type of a function definition

(types)	τ	::=	K	(primitive types)
		|	t	(type variables)
		|	$\langle\tau\rangle_{\times}$	(cart. prod. types)
		|	$\tau \to \tau$	(function types)
		|	$\tau[\langle\tau\rangle_{,}]$	(type applications)
		|	$\exists\, \vec{\alpha}.\ \tau$	(existential type)
(type schemes)	σ	::=	$[\forall\langle t\rangle_{,}]\tau$	
(expressions)	e	::=	k [: K]	(primitive constants)
		::=	v [: τ]	(variables)
		::=	$e(\langle e\rangle_{,})$	(function appl.)
		::=	$\langle e\rangle_{;}$	(expression sequence)
		::=	$\langle e\rangle_{,}$	(expression list)
		::=	$(c : \tau)(\langle e\rangle_{,})$	(constructor appls.)
		::=	(**fn**|**prc**) *rule*	(anonymous abstrs.)
		::=	**let** $\langle d\rangle_{;}$ **in** e **end**	(local declarations)
		::=	(**return**|**raise**|**exit**) [e]	(ret., raise exn., exit)
		::=	**if** e **then** e **else** e **end if**	(if-then-else)
		::=	**while** e **do** e **end while**	(while loop)
		::=	**begin** e **end**	(expr. sequence)
		::=	**pack** $\langle id = \tau\rangle_{,}$ **in** e **end pack**	(packing)
		::=	**unpack** e **as** e **in** e **end unpack**	(unpacking)
		::=	**case** e **of** $\langle rule\rangle_{\|\|}$ **end case**	(destructuring)
(rules)	*rule*	::=	e [“|”e] => e	
(declarations)	d	::=	(**var**|**con**) $x : \tau$[“ :=” e]	(object decls.)
		::=	**type** t[“[”$\langle\tau\rangle_{,}$“]”] = τ	(type decls.)
		::=	**class** c“ =” *classExpr*	(class decls.)
		::=	**local** $\langle d\rangle_{;}$ **in** $\langle d\rangle_{;}$ **end**	(local decls.)
		::=	**overload** $id : \tau$	(overload decls.)
(alg. type decls.)	*atdec*	::=	**alt** $\{\langle c$[**of** $\langle\tau\rangle_{\times}$]$\rangle_{,}\}$	

Figure 5.1: Simplified abstract syntax of Griffin

can often be left out, and type inference will be employed to determine the function type statically.

Functions are first-class objects in Griffin and closure mechanism is also supported. The keyword **fn** (so is **prc**) is used to construct anonymous function (procedure) literals; the syntax for these is similar to function definitions except the function name is optional. Functions may be overloaded. To overload a built-in operator function of Griffin, simply prefix the function name with the keyword **op**.

5.1.3 Types

The type formers of Griffin consists of *enum*, *alt*, *thread*, *channel*, and *rec*.

Union types (existential types)

A union type specifies a range of types, all belonging to the same class. Griffin's *union types* are essentially *bounded existential types*, existential types with possible constraints imposed on them. By analogy with universal quantification, the meaning of *existential types* (or *existentially quantified types*) is for any type expression $Q(\alpha)$,

$$x \ : \ \exists\alpha.Q(\alpha)$$

means for some type α, x has the type $Q(\alpha)$. The most general form of existential types is $\exists\alpha.\tau$. Not all existential types turn out to be useful. For example, if we have an object of type $\exists\alpha.\alpha$, we have absolutely no way of manipulating it (except passing it around) because we know nothing about it. The real usefulness of existential types becomes apparent only when we realize that modules (or packages) containing simple values become first-class citizens if they are modelled by existential types.

(Examples of such modules can be found in the language SOL developed by Mitchell and Plotkin [62]. However, severe restrictions exist when using existential types to model modules containing types. To overcome these obstacles, a more general notion such as *dependent types* [8] must be employed. MacQueen [54] introduced Σ-*closed structures* that successfully address the issues involved.) For instance, $\exists\alpha.\alpha \times (\alpha \to \texttt{int})$ is a simple example of an abstract data type packaged with its set of operations. The variable α is the abstract type itself, which hides an implementation. Existential types provide a type-theoretic account of abstract data types. For example, the data abstraction mechanism in Ada, Ada packages, can be described by records with function components.

Universal quantification yields generic types while existential quantification yields abstract data types. When these two notions are combined we obtain *parametric data abstraction.*

The existential types may be bounded in Griffin by restricting the types to a certain class. *Type casing* of objects of union types is not supported in Griffin so there is no need for an Griffin implementation to carry type tags at run-time.

The type of the well-formed Griffin expression

$$\textbf{pack}\ \ \alpha = \tau\ \ \textbf{in}\ \ M : \tau'\ \ \textbf{end pack}$$

is $\exists\alpha.\tau'$ according to the following type rule for packing (or wrapping, closing) expression:

$$\frac{\Gamma\ \rhd\ M : [\tau/\alpha]\tau'}{\Gamma\ \rhd\ (\textbf{pack}\ \ \alpha = \tau\ \ \textbf{in}\ \ M : \tau'\ \ \textbf{end pack})\ \ :\ \exists\alpha.\tau'} \qquad (\exists\textbf{-Introduction})$$

The operation **pack** is the only mechanism in Griffin for creating values of an existential type. M is referred to as the content of the existentially-typed value, whereas the type τ' is the *interface*. The interface determines the structural specification of the content and corresponds to the contract of a data abstraction. The *binding* $\alpha = \tau$ is the type representation: it binds the type variable α to a particular representation τ, thus corresponds to the hidden implementation associated with a data abstraction.

The following unpacking (or unwrapping, opening) rule makes available the type implementation:

$$\frac{\Gamma \ \rhd \ M : \exists\alpha.\tau \qquad \Gamma,\, p : \tau \ \rhd \ N : \tau'}{\Gamma \ \rhd \ (\textbf{unpack} \ \ M \ \ \textbf{as} \ \ p : \tau \ \ \textbf{in} \ \ N \ \ \textbf{end unpack}) : \tau'} \qquad (\exists\textbf{-Elimination})$$

wherein α is not free in τ'. Unpacking an object M (or package) of some existential type introduces a name p for the content of the object which can be used in the scope following **in**. As we do not know the actual definition of α (we only know that there is one), we cannot make assumptions about it, and users of objects of type α will be unable to take advantage of any particular implementation of it. The type of N cannot involve α to prevent the escape of the existentially-quantified type variable.

There is no subtyping relation among existential types; i.e., no two existential types, such as

$$\exists\alpha, \beta \,.\, \alpha \to \beta \qquad \text{and}$$
$$\exists\alpha \,.\, \alpha \to \alpha$$

are related by the subtyping relation.

Record type

Record type in Griffin, also called *abstract data type* (ADT), is an extension of the record type notation in most main-stream languages. The concept of ADT is better postponed until after the introduction of Griffin class; more detailed description of it can be found in Section 5.1.4.

Enum type

An example of Griffin's enum type:

```
type day = {Mon,Tue,Wed,Thu,Fri,Sat,Sun};
```

Thread type

In many applications it is natural to write a program as several parallel activities which synchronize as necessary. In Griffin, parallel activities are described by means of threads and channels (Section 5.1.6). Below is a definition of a Griffin thread type, which indicates the type that the thread returns when it completes. Griffin thread types are similar to Ada task types except that a return type is associated with threads; threads are essentially *task expressions*.

```
type intThread = thread[int];
```

Threads are created by using *thread expressions* and we are able to define thread objects just like any other objects. For example,

```
con t : intThread = thread(1+f(x))
```

creates a new thread to evaluate `1+f(x)` concurrently with the current thread.

Channel type

Channels are first-class objects in Griffin. A channel's type indicates whether the channel is synchronous or asynchronous and also determines the types of the values being transmitted. For example,

```
var ach : channel[int];
var sch : channel[int,real];
```

declares two channels, the first being an asynchronous channel and the latter synchronous.

Alt types

Algebraic types can be defined in Griffin with the keyword **alt** as follows:

```
type inttree = alt {empty, leaf(int), node(inttree; inttree)};
```

Other aggregate types

Tuples, sets, maps, lists need not be primitive; many of them can be defined in terms of others. In particular, the mechanism provided for defining ADT in Griffin is powerful enough to define most of the above types.

SETL, with its use of sets and maps, is regarded as a vehicle for rapid experimentation with algorithms and program design. Unlike SETL, aggregates in Griffin are homogeneous. Griffin provides a concise syntax for constructing and iterating over specific forms of aggregates, for details see Section 5.1.7 and 7.12.

5.1.4 Type system

A *type system* [21, 20, 80] is a collection of rules for assigning type expressions to various parts (mostly expressions) of a program. A *type checker* implements a type system. Griffin's type system is an extension of the Hindley-Milner [57, 26, 69] polymorphic type system; such systems allow functions accept not only parameters of ground types, but also parameter of any type.

Griffin has been designed trying to combine the safety of compile-time type checking with the flexibility of *declaration-less* programming by inferring type information from the program rather than insisting on extensive declarations.

A type definition in Griffin introduces either a *generative* or a *non-generative* type. A generative type definition defines a new type which is distinct from all other existing types. A non-generative type definition associates a name with a type expression. An **alt** type definition in Griffin introduces a generative type. The rules for both generative and non-generative types are given in Figure 5.2

Polymorphism

Polymorphism is attractive because it facilitates the implementation of algorithms that manipulate data regardless of the types of them. Griffin supports polymorphic types — types that are universally quantified in some way over all types. Parametric polymorphism is a special kind of polymorphism in which type expressions can be parameterized.

A polymorphic function can be applied to arguments of different types. As an example of an user-defined polymorphic function that operates on lists, consider the problem of counting the number of elements in a list:

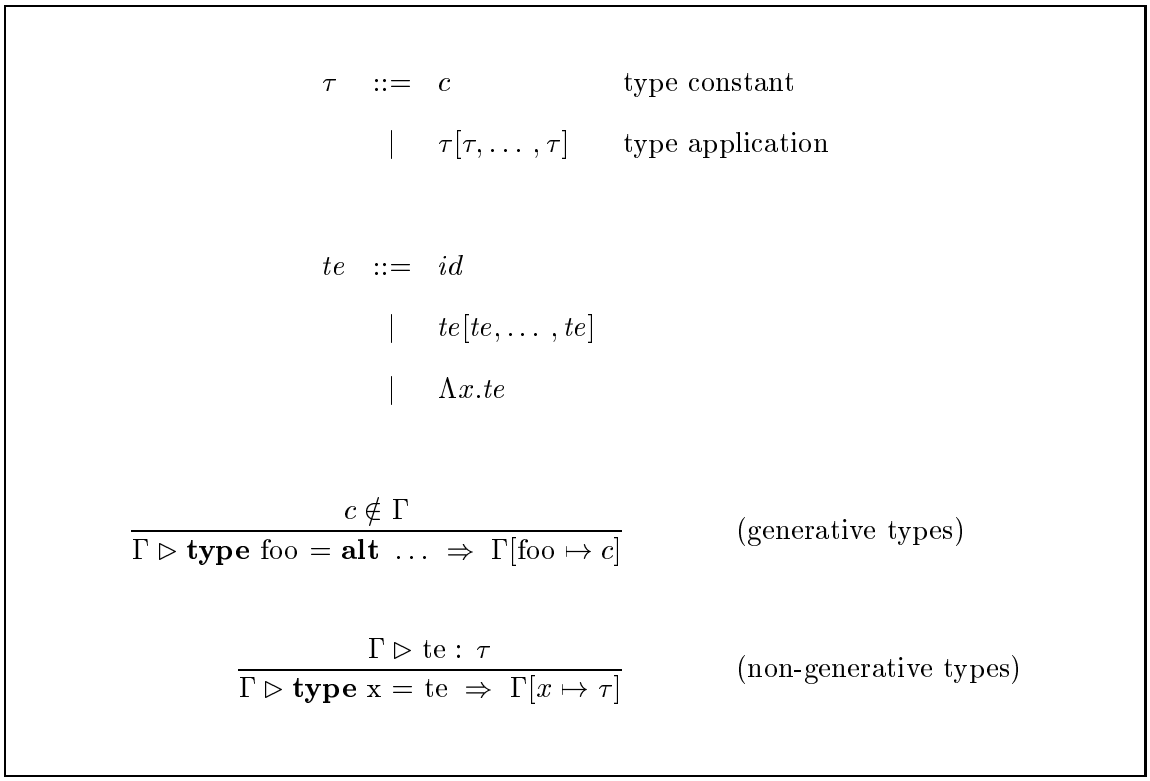


Figure 5.2: Type rules for generative and non-generative types

```
fun length([]) => 0;
 |  length(x^xs) => 1+length(xs);
```

where ^ is the Griffin predefined list constructor. The definition is almost self-explanatory: the length of the empty list is 0, and the length of a list whose first element is `x` and remainder is `xs` is 1 plus the length of `xs`.

Bounded polymorphism and classes

Unconstrained polymorphism is too liberal in some sense; a polymorphic function knows nothing about the structure of its parameters, nor what operations can be

applied to them. In *bounded polymorphism*, type variables are constrained to range over a certain domain, as opposed to in unconstrained universal polymorphism where type variables can be instantiated to any type. A simple example of bounded polymorphism is the *equality* problem. There are many types for which we would like equality defined, but some for which we would not. For example, comparing the equality of functions is undecidable, whereas we often want to compare two lists for equality. To highlight the issue, consider this definition of the infix operator function `in` which tests for membership in a list:

```
fun x op in ([]) => false
 |  x op in (y^ys) => (x=y) or (x in ys)
```

Recall that `^` is the list constructor. For stylistic reasons, operators like `in` are defined in infix form. Intuitively speaking, the type of `in` should be: $\alpha \rightarrow list\ \alpha \rightarrow bool$. But this would imply that `=` has type $\alpha \rightarrow \alpha \rightarrow bool$, though we just mentioned we do not expect `=` to be defined for all types. Even if `=` were defined on all types, comparing two lists for equality is very different from comparing two integers. In this sense, `=` is expected to be overloaded to carry out these various tasks.

In Griffin, *classes* provide a structured way to control *ad hoc polymorphism* (or *parametric overloading*) [43]. Classes conveniently fixes both problems by allowing us to declare which types are *instances* of a certain class, and to provide definitions of the overloaded operations associated with a class. For example,

```
class Eq = {fun op = (mytype;mytype) : bool;};
```

Class `Eq` denotes the set of all types such that there is an operator `=` defined which takes two parameters of that type and returns a value of boolean type.

We can now use the `Eq` class to refine the function definition of `in` mentioned above:

```
fun [t::Eq] (x:t) op in ([]:list[t]) => false
 |  x op in (y^ys) => (x=y) or (x in ys)
```

The parameter can be of any type `t` that is in the set designated by class `Eq`. The polymorphism of the function `in` is suitably bounded by the class mechanism.

Abstract data types

Abstract data type (ADT) supports the *information hiding principles* and control access. Information hiding principles refers to that one must provide the intended user with all the information needed to use the module correctly and *nothing more*; the implementor is furnished with all the information needed to complete the module and *nothing more*.

ADT is broken into two parts – an *interface specification* and a *body*. The interface specification defines the interface between the inside and the outside of the ADT; it is effectively a *contract* between the user and the implementor of the ADT. The body may contain local functions (helper functions), variables, types needed by the implementation.

In Griffin, the class facility seen above is used to specify the interface of an ADT, and the body of an ADT appears in the definition of the type.

Parameterized classes and types

Griffin classes and types can be parameterized by other types (including parameterized types) for richer expressiveness. Parameterized types (classes) are derived by

applying type (class) constructors to appropriately constrained type parameters. It is desirable to have a sufficiently powerful type system in a programming language to facilitate the definitions of common aggregate types; parameterized types and classes turn out to be the instruments for such purpose in Griffin.

We should mention that there are problems in deciding, in general, when two parameterized recursive type definitions represent the same type. Marvin Solomon [80] described the problem and a reasonable solution which involves restricting the form of parametric type definitions. Griffin's type system observes these restrictions to avoid the semi-decidability problem.

5.1.5 Type equivalence

Griffin uses structural equivalence for all types except when the keyword **new** is used in generative type definitions. Sometimes it is useful to introduce a new type which is similar in most respects to an existing type but which is nevertheless a distinct type. If `t1` is a type,

```
type t2 = new t1;
```

introduces a new type `t2` with the same operations as `t1` but there are no interoperations between `t1` and `t2`.

5.1.6 Concurrency

Concurrency in Griffin [3] is achieved via the use of *threads* and *channels* which are generalization of Ada tasks and entries. Threads provide concurrent execution while channels provide communication and synchronization between threads.

Thread variables are declared in the same manner as any other variables. Following is an example of thread creation by using thread expressions:

```
type intThread = thread[int];
var t: intThread = thread(1+f(x));
```

in which a new thread, `t`, is created to evaluate `1+f(x)` concurrently with the current thread.

There are several predefined operations for thread in Griffin, including `status(t)`, which indicates if `t` is still being executed, `abort(t)`, which causes `t` to be aborted, and `value(t)` which blocks until `t` is finished and then returns `t`'s associated value.

Griffin threads communicate via channels. Channels are passive concurrent-access data structures that provide bidirectional anonymous communication between multiple senders and multiple receivers. Channels may be either synchronous or asynchronous. Both threads and channels are first-class objects which can be passed as parameters, returned as values.

Here are some examples of sending and receiving on channels:

```
var ach : channel[int];
var sch : channel[int,real];
send(ach,1);
recv ach(t) => foo(t); end;
r1 := send(sch,1);
r2 := sch(2);              -- just like an Ada entry call
recv ach(i) => i+3.0; end;
```

The variable `ach` is a asynchronous channel variable of type `channel[int]`, on

which nonblocking **send** and **recv** can be performed. The next declaration creates a synchronous channel, `sch`, over which integers are sent. The **recv** statement waits for a message to be sent to a channel and then executes the body of the **recv**. The statements to be obeyed during a rendezvous are described in the body of **recv**, between `=>` and the keyword **end**. After the **end** there may be further statements; they are executed outside the critical region.

Griffin provides a **select** statement which allows a thread to wait for a message to arrive at one of several possible channels. A more general form of **select** statement includes the use of the guarding conditions. Each branch of the **select** statement commences with

when `condition =>` ...

and is then followed by a **recv** statement. Each time the **select** statement is encountered all the guarding conditions are evaluated. The behavior is then as for a **select** statement without guards but containing only those branches for which the conditions were true.

5.1.7 Iterators, generators, and comprehension expressions

In this section, we give a formal introduction to comprehension expressions, which will be used in later discussion (Section 7.12). The generalization of aggregate (set, map, bag, list, etc.) comprehension is *monoid comprehension* [17]. A Griffin generator can be depicted as a *monoid homomorphism* [16]. In universal algebra a *semigroup* is a set $\mathbb{U}$ together with an associative binary operation $\oplus : \mathbb{U} \times \mathbb{U} \to \mathbb{U}$. The set $\mathbb{U}$ is called the *universe* (short for *universe of discourse* in formal logic) or *carrier set* of the semigroup. A monoid is a semigroup with an identity element;

thus a monoid is not allowed to be empty: it must contain at least an identity element. If $\mathcal{S}$ and $\mathcal{T}$ are semigroups with binary operation $\oplus$ and $\otimes$ respectively, a function $h : \mathcal{S} \to \mathcal{T}$ is a homomorphism between $\mathcal{S}$ and $\mathcal{T}$ if for all $s, s' \in \mathcal{S}$, $h(s \oplus s') = h(s) \otimes h(s')$. Homomorphism between monoids preserves the identity elements: if e is the identity element of $\mathcal{S}$, $h(e)$ must be the identity element of $\mathcal{T}$.

The aggregate *generator* is used to specify that a certain calculation is to be performed for each of the elements in a given aggregate object. Common examples includes higher-order functions *map* and various *filter* functions.

An example helps illustrate the relationship between Griffin generators and homomorphism between monoids. Consider the problem of summing all the elements of an integer set. In SETL, the expression `/+S` denotes the sum of all the elements of set `S`, where / is a *reduction* operator. Following the notation used above, the set expression $\{a_1, \ldots, a_n\}$ can be modelled with the monoid $\mathcal{S}$ defined in Figure 5.3, where $\cup$ (the binary operator) is the set union operator and $\mathbb{N}$ (the universe) is the set of integers. The set itself can be apprehended constructively as $a_1 \oplus (a_2 \oplus \ldots (a_n \oplus e) \ldots)$, in which $\oplus$ is $\cup$. The reduction operator `/+` is an homomorphism mapping monoid $\mathcal{S}$ to the monoid $\mathcal{T}$ (also defined in Figure 5.3). Using the notations above, we derive:

$$\begin{array}{lll} & \{a_1, \ldots, a_n\} & = \; a_1 \oplus (a_2 \oplus \ldots (a_n \oplus e) \ldots) \\ \overset{/+}{\Longrightarrow} & a_1 \otimes (a_2 \otimes \ldots (a_n \otimes h(e)) \ldots) & = \; a_1 +_{\mathbb{N}} \ldots +_{\mathbb{N}} a_n \end{array}$$

The viewpoint is similar to that of the abstract list traversal operations `foldleft` and `foldright` in ML. ML's `foldleft` and `foldright` only work on of lists. Griffin generators can be defined for all aggregate types to allow concise algorithmic programming.

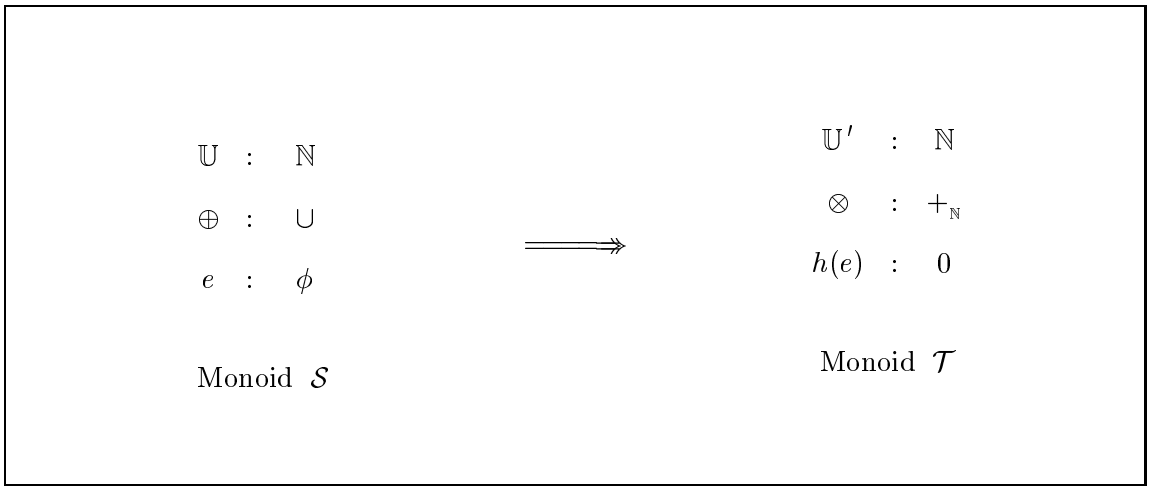


Figure 5.3: Monoid homomorphism

Generators are used in many Griffin constructs. For example, they are used in aggregate comprehension expressions to form other aggregates, in loop expressions, or combining with quantifiers to build useful queries.

A common query in programming contexts is: do all the objects in an aggregate satisfy some stated criterion? Queries of this kind would be expressed in Griffin by means of constructs such as the following:

forall `x` **in** `s1, y` **in** `s2, z` **in** `s3 | guardCondExpr`

The guard expression `guardCondExpr` is used to test each expression as generated, with the generator terminating yielding false when a false value is encountered (in this case, `x`, `y` and `z` get the value undefined); otherwise yielding the value true.

Similarly, in

exists `x` **in** `s1, y` **in** `s2, z` **in** `s3 | guardCondExpr`

the construct searches the aggregate of all `x in s1`, `y in s2`, `z in s3` for values satisfying the guard condition *guardCondExpr*. If any such values are found, then

it yields the value true and the variables x, y and z are instantiated to appropriate values; otherwise it returns false and leaves x, y and z undefined.

In fact, Griffin generators are more general than what is mentioned above: they do not have to be associated with aggregates. For example, the generator that produces the first 50 even integers is:

```
{ i : 1..100 | even(i) }
```

where even is a boolean-valued predicate with a single integer parameter.

5.1.8 Exceptions

Exception is a mechanism to handle deviant conditions, so errors can be signalled and trapped. When an exception is raised in Griffin, it is transmitted by all functions on the stacks until it is *caught* by an exception handler. An exception handler tests for particular errors by pattern-matching so it is similar to a case expression. Unlike Ada, Griffin exceptions can return values. The exception handler specifies what to return for each kind of exception. An exception name in Griffin is a constructor of the built-in type exn. There is one major difference between a case expression and a exception handler: if no pattern matches, then the handler propagates the exception rather than raise the built-in exception `Pattern matching failure`. Exceptions can be viewed as objects of a datatype with a unique property: its set of constructors can be extended.

5.1.9 Pattern matching

In Griffin, pattern matching [33, 44, 46] is used in function definitions, lambda expressions, bindings, and case expressions. Patterns are linear in Griffin. Because

Griffin allows user-defined deconstructors, patterns may overlap, meaning that more than one pattern can match the same subject (or subject expression). In case ambiguity occurs we impose ordering — the order in which the patterns are written down to decide which rule to apply. Shown in Figure 5.4 is an example that illustrates the overlapping resulting from user-defined deconstructors.

Griffin allows the layered pattern syntactic constructs, which are of the form `v` **as** `P` where `v` is an identifier and `P` is a pattern. As far as matching is concerned, it is equivalent to `P`. To implement the layered pattern the only extra work is to update the environment by binding `v` to `p` if the pattern matching succeeds.

```
type t = alt {a,b,c};

fun ~ab(x) =
   if x=a or x=b
   then ...
   else ...

case exp of
   ab  =>  "ab"  ||
   a   =>  "a"   ||
   b   =>  "b"
```

Figure 5.4: Overlapping pattern in Griffin

In Griffin, a function definition defined by a set of pattern matching equations is as follows:

fun `foo` P_0 `=>` E_0`;`
| `foo` P_1 `=>` E_1`;`
⋮
| `foo` P_{n-1} `=>` E_{n-1}`;`

5.1.10 Extensible records

Practical languages are less flexible in the operations for record manipulations. For example, they do not support *polymorphic* operations on records — such as a general field selector function that extracts a value from any record that has an field l.

In ML, the polymorphic function

```
fun foo r = #l r;
```

is not allowed because it assumes a *flexible record* and its principal typing scheme requires consistent treatment of polymorphic functions.

Wand [90] introduced the concept of *row variables* to allow incremental construction of record types. Rows are of the form $\{|l_1 : \tau_1, \ \ldots \ , \ l_n : \tau_n|\}$ where l_i $'s$ are labels and τ_i $'s$ are their associated types; empty record is denoted by $\{|\ |\}$. Record types are obtained by preceding rows with the symbol *Rec* and variant types are constructed using *Var*; the situation is similar between these two as far as type system is concerned so we only address the issues of records hereafter. Intuitively, a record of type $Rec\{|l : \alpha \,|\, r|\}$ is like a pair whose first component is a value of type α, and whose second component is a value of type $Rec\, r$. Repeated use of any label within a particular row is disallowed. According to the formulation above the following two type expressions denote the same type

$$\{|l_1 : \tau_1, \ \ldots \ , l_n : \tau_n|\} = \{|l_1 : \tau_1 \ \mid \ \ldots \ \mid \ \{|l_n : \tau_n|\} \ \mid \ \{|\ |\}\ |\}$$

Griffin record extension is based on the work described in [38] where the authors present a way to support polymorphism and extensibility over record type, type

inference and compilation. Note that two record types are considered the same in Griffin if they include the same fields, regardless of the order in which those fields are listed.

5.2 Macrosemantics specification of Griffin

The complete macrosemantic description of Griffin is given in Appendix A. The description is processed by a macrosemantics processor, which yields a static analyzer for Griffin in ML. The majority of the macrosemantic description comprises of semantic equations of the form

$$\begin{aligned} &\mathit{semFunName} \quad [\![\ \mathit{syntacticConstruct}\ ]\!] \quad \mathit{otherParams} \\ &\quad = \quad \mathit{semEqnRHS} \end{aligned}$$

where *semFunName* is the name of the current semantic equation, *syntacticConstruct* is the syntactic construct defined by the semantic equation, *otherParams* denotes additional parameters (environment, etc.) of the function, *semEqnRHS* is the right-hand-side of the semantic equation containing information about the static analysis of *syntacticConstruct*, the actions of the components of *syntacticConstruct*, and the action representing the current syntactic construct.

Note that the type system used in G2A is parameterizable, i.e., we have separated the static analyzer from the type system in such a way that it is possible to plug in different type systems as long as they satisfy the requirements laid down by the interface for the static analyzer. This is a useful feature for experimenting with various type systems especially in the development stages of a prototyping language.

Chapter 6

Overview of Ada95 as a target language

The programming language Ada (after Ada Augusta, the Countess of Lovelace) is a high-level programming language originally sponsored by the U.S. Department of Defense for use in the so-called embedded system application area. First published in 1980, the unconventional part of Ada includes concepts such as data encapsulation, exception handling, concurrency, and generic subprograms. The conventional part of Ada includes concepts present in then existing high-level languages such as Pascal, C, PL/I, and Fortran.

Ada95 [1, 2, 7] is the revised version of it designed and standardized to support and strongly encourage widely recognized software engineering principles: reliability, portability, modularity, reusability, efficiency, maintainability, information hiding, abstract data types, genericity, concurrent programming, object-oriented programming, etc.

A brief introduction to Ada and Ada95 are given in this chapter. There is considerable semantic gap between the source language Griffin and target language Ada95 that we experiment with in this thesis work. In the following sections we will address certain features in Griffin that do not translate straightforwardly to Ada.

Although high performance is not our primary goal, the performance of the code generated should compare favorably with the output of other language translators by carefully choosing the semantic algebra, which implements the operators in the action notation terms. Various optimization techniques can be applied to the target code through a term-rewriting system to further improve the efficiency.

6.1 Ada83

Ada (often referred to as Ada83 nowadays) will continue in use in its own right for many years. Ada95 has been designed so that the great majority of Ada83 will behave identically as their Ada95 programs to maximize compatibility.

6.1.1 The traditional language constructs

We will briefly discuss the traditional programming language aspects of Ada83 in this section.

Type declarations

A type in Ada is either an elementary type or a composite type. Elementary types cannot be decomposed further whereas composite types are composed of a few of components. Ada provides standard elementary types for integers, floating

point numbers, characters and booleans. In addition, it provide four kinds of user-defined types: enumeration types, array types, record types and pointer types (called **access** types in Ada). There are predefined functions that apply to these types.

Ada adopts name equivalence to determine the equivalence of type expressions (see Section 7.1 for more details).

Data declarations

A type is a set of values plus a set of operations that can be performed upon these values. An object is an entity with which a type is associated; a value of this type can also be associated with the object. An object is created and its type specified by means of a declaration. All objects must be explicitly declared in Ada. An initial value may be given to the object in the declaration (which will override a default initial value associated with the object type). There are two kinds of objects — constants and objects. The value given to a constant cannot be changed, while that given to a variable can be changes.

Expressions

Expressions are formed using operators and operands. In evaluating an expression, operators with a higher precedence are applied first. Operators having the same precedence are applied in textual order from left to right. Parentheses may be used to change the order of evaluation imposed by the precedence of the operators.

Control statements

The control statements found in Ada are of the following kinds: assignment statements, procedure statements, conditional statements, iteration statements, and control flow statements. In addition to permitting assignment to structured variables, Ada allows assignments to parts of arrays (called slices).

Subprograms and parameters

Subprograms in Ada come in two varieties — procedures and functions. A procedure is executed for its side effect (e.g., changing the values of the **in out** parameters, supplying values to **out** parameters or updating global variables) and functions are used to return values.

Subprograms are invoked (executed) by means of subprogram calls. A procedure call is a statement, while a function call is an operand in an expression. Functions in Ada can only have **in** parameters. Ada supports name parameter association, which enables the programmer to disregard the order of subprogram parameters and explicitly associates a particular value with the desired parameter.

Scoping rules

In Ada, visibility of names comes from the current and outer scopes. The scope rules for blocks are identical to those for procedures and functions. The naming convention for task entries is the same as that for record fields: a particular entry is selected by prefixing its name with the task name.

6.1.2 Language support for software engineering issues

Software engineering is concerned with the design, organization, implementation and maintenance of large systems. The main concern of software engineering is the complexity that results from the largeness of these systems.

Modularity

For a large software system, modularity support from the language in order to partition the system into separately manageable pieces is indispensable. Ada supports this need in two ways: by program unit constructs and separate compilation. Besides the traditional modularization constructs of subprograms (i.e., procedures and functions), Ada provides the task and package constructs. The latter is particularly important because it enables the designer to group related program pieces into larger units.

A package consists of two distinct parts: the visible specification and the implementation body. The visible part is accessible to the user of the package and may contain declarations of types, constants, data objects, subprograms, tasks, and even packages. The implementation, however, is hidden from the user.

Concurrency

Systems that require concurrent processes are very difficult to write correctly in languages that do not support concurrency. Problems arise in areas such as access to shared data and sharing resources. Ada was one of the few programming languages that provides support for concurrent processes in the language itself in early 1980s. The construct supporting concurrency in Ada is the task construct. Structurally

a task is similar to a package: it consists of a visible part and an implementation body. Notwithstanding the similarities between tasks and packages, there are two basic differences: in contrast to a package body, a task body defines an independent activity and the task interface defines a number of entry points in the task.

Exception handling

A significant part of a large software system is concerned with handling errors. The Ada language was designed for programs whose reliability is essential. Such programs must be capable of responding in a sensible way to unexpected situations. Depending on the application, this sensible response may entail terminating in a well-defined state, issuing a warning and continuing normal execution, retrying some computation using a different algorithm, or continuing execution in a degraded mode, for example.

In the Ada language, an unexpected situation is called an exception. The response to an exception is specified by a handler for that exception. Handlers can be specified at the end of subprogram body.

6.2 The target language: Ada95

Among the new additions of Ada95, *protected types* are particularly useful in the translation of Griffin threads and channels. Ada95 introduces a low overhead, data-oriented synchronization mechanism based on the concept of protected objects. The operations on a protected object allow two or more tasks to synchronize their manipulations of shared data structures. From the implementation perspective, a protected object is designed to be a very efficient conditional critical region. The

protected operations are automatically synchronized to allow only one writer or multiple readers. The protected operation are defined using a syntax similar to a normal subprogram, with the mutual exclusion of the critical region happening automatically on entry, and being released automatically on exit.

6.2.1 Object-oriented features

A brief introduction to the object-oriented paradigm is first given in this section, followed by a quick overview of the object-oriented features in Ada95. Unless otherwise stated, *class* in this section refers to the concept used in object-oriented paradigm rather than Griffin's class. Over the past two decades, the essential concepts of object-oriented paradigm, namely inheritance and polymorphism, have emerged as mechanisms that can guarantee interface compatibility at compile time while deferring the binding to particular types or subprogram implementations to run time. A major advantage of this approach is the reuse of existing reliable software without the need for modification, recompilation, and retesting.

In object-oriented design and programming, the most fundamental concept is that the program is a model of some aspects of reality. The *classes* in a program represent the essential notion of the "reality" being modelled. Real-world objects and artifacts of implementation used by the designers and programmers are represented by objects of these classes.

The concept of derived classes and its associated language mechanism is to express hierarchical relationship, i.e., the commonality between classes. Derived classes provide a simple, extensible, and efficient mechanism for defining a class by adding facilities to an existing class without reprogramming or recompilation.

Inheritance is the sharing of attributes and operations among classes of objects based on a hierarchical relationship. *Generalization* is the relationship between a class and the refined versions of it. Generalization and inheritance are powerful abstractions for sharing similarities among classes while preserving their differences. The class being refined is called the *superclass* and each refined version is called a *subclass*. Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass. Each subclass is said to *inherit* the features of its superclass. Generalization and inheritance are transitive across arbitrary number of levels. A superclass is sometimes called a *base class* or a *parent class*, and a subclass a *derived class* or a *child class*. A subclass *extends* its superclass and thus can be used in places where the superclass is legal.

This is a form of polymorphism: an object of a given class can have multiple forms, either as a member of its own class or any superclass it extends. The run-time choice of functions (procedures) taking argument belonging to a superclass is called *dynamic dispatching* and is key to the flexibility of class-wide programming.

A class can be defined broadly and then refined into successively finer subclasses. Each subclass incorporate, or inherits, all of the properties of its superclass and adds its own unique properties. The properties of the superclass need not be repeated in each subclass. The ability to factor out common properties of several classes into a common superclass and to inherit the properties from the superclass can greatly reduce the repetition within design and programs and is one of the main advantages of an object-oriented system. Inheritance has become synonymous with code reuse within the object-oriented programming community. Often code is available from past work (such as a library) which the developer can reuse and modify, where

necessary, to get the precise desired behavior.

Type extension of Ada83 builds upon the concept of *derived types*. A derived type inherits the operations of its parents and could add new operations but not new components. The whole mechanism is thus somewhat static. By contrast, Ada95 allows extension to a type by adding new components, thus it becomes much more dynamic and flexible.

In Ada95, tagged types are record types which may be extended on derivation. As the name implies, values of tagged types carry a tag at run time. With each tagged type `T` there is an associated class wide type `T'class`, whose values are the values of `T` and all its derived types. A subprogram that takes a class-wide argument cannot know of the specific types because it needs to work if a new tagged type is added to the class. This runtime choice of subprograms is the key to achieve dynamic dispatching in Ada95.

Some languages permit a subclass to be derived from more than one superclass. These languages are said to support *multiple inheritance*. Ada95 does not support the general form of multiple inheritance. However, some forms of multiple inheritance can be simulated with existing language features.

Multiple dispatching refers to that the dynamic dispatching can depend on more than one argument. To avoid run-time inefficiency and adding another dimension of complexity to the language semantics, multiple dispatching is not supported in Ada95 either.

6.2.2 Type system

Tagged records in Ada95 are generalized record types which can be extended and form the basis for object-oriented programming. A class-wide type is declared implicitly whenever a tagged record type is defined. Protected types are composite types that provide synchronized access to the shared data components via a number of protected operations. Objects of protected types are passive and do not have a distinct thread of control; the mutual exclusion is provided automatically.

Chapter 7

Target code generation

In this chapter we outline the translation techniques used to deal with some of the semantic differences between Griffin and Ada95, such as structural versus name equivalence of type expressions, garbage collection, closure conversion, exception, and concurrent constructs. The translations of certain Griffin features such as iterators, generators, comprehension expressions, and pattern matching that are not straightforward are detailed in the form of macrosemantic equations.

7.1 Structural equivalence versus name equivalence

Depending on the treatment of names, two divergent concepts of equivalence of type expressions arise. *Name equivalence* views each type name as a unique type. Under *structural equivalence*, names are replaced by their definitions, so two type expressions are considered structurally equivalent if, after all type names have been substituted out, they represent two structurally equivalent type expressions. For

example, in the following declaration

$$\textbf{type } \texttt{t} = \tau$$

`t` denotes a new type distinct from any other type in name equivalence, whereas under structural equivalence, it is merely a synonym of the type expression τ.

For convenience of programmers, structural equivalence is superior in some cases. On the other hand, name equivalence simplifies the decidability and complexity issues of a type system. Therefore, even though most programming languages primarily have either structural equivalence or name equivalence, a hybrid form of both notions is implemented for practical considerations.

Griffin adopts structural equivalence while Ada uses name equivalence. Let $\tau_1 \equiv \tau_2$ denote the type expression τ_1 is structurally equivalent to τ_2. Note that in Griffin, two *recursive types* [61] are never structurally equivalent, because the only way to introduce a recursive type is through the algebraic type former **alt**, which always results in a generative type. For basic type expressions, substitutions of type expressions for type names is enough to check whether two type expressions are structurally equivalent or not. It is then straightforward to inductively verify the structural equivalence of compound type expressions.

G2A performs whole-program transformation, i.e., it needs all the Griffin program components available before the translation process starts. We first collect all the structurally equivalent type expressions, $\tau_1, \ldots, \tau_n$, in the Griffin program. Since structural equivalence induces an equivalence relation among type expressions,

$$\tau_1 \equiv \ldots \equiv \tau_n$$

implies

$$[\tau_1] \;=\; \ldots \;=\; [\tau_n]$$

where $[\tau]$ designates the equivalence class (or the "type", "abstract type") of the type expression τ. A unique type name `t` is introduced in the target code (most likely to be wrapped in a Ada package) for each equivalence class, and all occurrences of $\tau_1, \ldots, \tau_n$ in the source programs are replaced with the type name `t` in the target program.

Note that after this stage of transformation, each type check only takes constant time in the target code.

7.2 Coercion calculus based translation of existential types

Existential types [61, 20] can be used to model implementations of abstract data types. An existential type asserts the existence of objects that have a particular instance of the type, without revealing the type information of the particular instance, thereby making the usage of the existential type independent of any particular instance type.

A *coercion* [60, 41, 40] is a conversion from one type to another, which can be explicit or implicit. According to [4], coercions are limited in many languages to situations where no information is lost in principle; e.g., an integer may be converted to a real but not vice-versa. In practice, however, loss is possible when a real number must fit into the same number of bits as an integer.

Two primitive operations in Griffin are used to construct and deconstruct objects of an existential types: packing (or wrapping, closing) introduces the existential

quantifier $\exists$ into the type of objects, and unpacking (or unwrapping, opening) eliminates the existential quantifier $\exists$. Operationally, these two operations do not actually change the value of an object; they behave like type coercions.

According to the Griffin syntax given in Section 5.1.1, packing and unpacking have the following form:

$$\textbf{pack } \langle id = \tau \rangle_{,} \textbf{ in } e \textbf{ end pack}$$

$$\textbf{unpack } e \textbf{ as } p \textbf{ in } e \textbf{ end unpack}$$

The operation **pack** is the only mechanism for creating objects of existential types in Griffin. Objects of an existential type must be unpacked before they can be used. In Griffin, **pack** and **unpack** are the only two primitive operations that can manipulate objects of existential types. A simple example demonstrating the use of existential types is given in Figure 7.7.

We will discuss the translation of existential types in their full generality, i.e., with an arbitrary number of $\exists$-quantified type variables, and nested at different levels. There are several possible choices for the translation; e.g., converting all uses of an existential type to a corresponding universal type appearing at the dual positions [40]. That is, a use of the existential type $\exists\alpha.Q[\alpha]$ such as $(\exists\alpha.Q[\alpha]) \rightarrow \beta$, where $\alpha \notin \mathit{freevar}(\beta)$ (so no escaping is possible), is converted to its equivalent form $\forall\,\tau.(Q[\tau] \rightarrow \beta)$. This equivalence can be intuitively understood from the syntax of Griffin **unpack** expression (the only way to manipulate objects of an existential type in Griffin)

$$\textbf{unpack } e_1 \textbf{ as } p \textbf{ in } e_2 \textbf{ end unpack}$$

where e_1, p, e_2 are of types $\exists\alpha.Q[\alpha]$, $Q[\alpha]$, and β, respectively. The part "**as** p **in** e_2" can be viewed as a function $\lambda p.e_2$, which takes an argument of $Q[\alpha]$ and

returns a value of type β, universally quantified over type variable α. The entire **unpack** expression is thus an application of this function to expression e_1, therefore of type $\forall\,\tau.(Q[\tau] \rightarrow \beta)$. In this translation scheme, objects of type $Q[\tau]$ which is an instance of $\exists\,\alpha.Q[\alpha]$ in the source program are mapped to objects of the type $\mathbb{T}[\![Q[\tau]]\!]$ in the target program, where $\mathbb{T}[\![Q[\tau]]\!]$ denotes the translation of $Q[\tau]$. The drawback of this approach, however, is that objects of a certain existential type in the source program are no longer of the same type in the target program. Therefore, it violates our goal of type-preserving translation—objects of the same existential type in Griffin should be translated into the same type in Ada95. Due to this problem, we explore an alternative approach by using the object-oriented features in Ada95 to give a type-preserving translation.

If Q is a term and θ is a *substitution*, θQ is a term obtained by applying substitution θ to Q. We treat substitutions as functions. For example, when substitution $\theta = [x \mapsto y]$ is applied to a term Q, θQ is the term obtained by substitutes y for all free occurrences of variable x in Q.

The general form of a Griffin existential type is $\exists\,\vec{\alpha}.Q[\vec{\alpha}]$, where $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$, and $Q[\vec{\alpha}]$ is a type expression with possible free occurrences of type variables $\alpha_1, \ldots, \alpha_n$. Essentially, for each Griffin existential type (modulo α-conversion) $\exists\vec{\alpha}.Q[\vec{\alpha}]$, where $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$, a corresponding vector of abstract base types $\vec{\mathcal{B}_\alpha} = (\mathcal{B}_{\alpha_1}, \ldots, \mathcal{B}_{\alpha_n})$ denoting all instances of $\vec{\alpha}$, is defined in Ada95. All objects of the type $\exists\vec{\alpha}.Q[\vec{\alpha}]$ in Griffin are mapped to objects of types $\mathbb{T}[\![[\vec{\alpha} \mapsto \vec{\mathcal{B}_\alpha}]Q]\!]$ (thus all of them have the same type in the target code), with the $\exists$ quantifier removed and α_i substituted by $\mathcal{B}_{\alpha_i}$, $\forall\, i \in 1 \ldots n$.

Given the syntax of the source and target types in Figures 7.1 and 7.2, we for-

mulate the type translation that eliminates the existential quantifier in Figure 7.3. Here, we assume that all type variables occurring in the source type expression are introduced by an existential quantifier, so we can translate them into the corresponding base type in the target program. For other type variables, since they remain unchanged while the packing and unpacking operations are applied, they can be simply be treated as type constants.

Before we proceed with the translation of existential types (along with the associated packing and unpacking operation), here we give the outline of deriving the translation:

- First, we need to determine the general syntactic relationship between the translations of two source types related by the packing/unpacking operations, i.e., an existential type $\exists\vec{\alpha}.Q[\vec{\alpha}]$ and an arbitrary instance $Q[\vec{\tau}]$ of it. We prove a suitable Substitution Lemma to show that the translation of the instance is a substituted form of the translation of the existential type. The problem of translating the packing and unpacking operations in the source program is then reduced to the problem of finding invertible coercions between these two target types.

- Next, we present a general approach to inductively construct coercions between translations of an existential type and its instances, given the base coercions between translations of the $\exists$-bound type variable and a ground type. Such base coercions are constructed using the class hierarchy in Ada95.

The scheme to translate existential types is illustrated in Figure 7.4. Here, without loss of generality, we concentrate on the case where the existential quantifier introduces only one type variable α (in the implementation, we use the more

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \text{(type variables)} \\
 & | & K_{\text{Grf}} & \text{(primitive types)} \\
 & | & \tau_1 \to \tau_2 & \text{(function types)} \\
 & | & \exists \vec{\alpha}.\tau & \text{(existential types)} \\
 & | & \tau_1 \times \cdots \times \tau_n & \text{(product types)}
\end{array}
$$

Figure 7.1: Griffin source types (with $\exists$ quantifiers)

$$
\begin{array}{rcll}
\tau & ::= & \mathcal{B}_\alpha & \text{(base types)} \\
 & | & K_{\text{Ada}} & \text{(primitive types)} \\
 & | & \tau_1 \to \tau_2 & \text{(function types)} \\
 & | & \tau_1 \times \cdots \times \tau_n & \text{(product types)}
\end{array}
$$

Figure 7.2: Ada95 target types (without $\exists$ quantifiers)

general form of existential types that quantifies over a vector of type variables as a notational convenience, since though type $\exists\vec{\alpha}.\tau$ is isomorphic to the type $\exists\alpha_1.\cdots.\exists\alpha_n.\tau$, the former form incurs less run-time cost than the latter because fewer packing/unpacking operations are needed). The type β, an arbitrary ground type, is wrapped by the packing operation. As usual, the substitution $[\beta \mapsto \alpha]\,\tau$ is non-capturing. The implementation essentially mimic the effect of packing and unpacking using a pair of coercion functions *embed* and *proj*. In order to do so, we need to determine the relationship between the two target types.

Lemma 7.1 (Substitution Lemma). *Given source type $\exists\alpha.\tau$, ground source type*

$$
\begin{aligned}
\mathbb{T}[\![\alpha]\!] &= \mathcal{B}_\alpha \\
\mathbb{T}[\![K_{\text{Grf}}]\!] &= K_{\text{Ada}} \\
\mathbb{T}[\![\tau_1 \to \tau_2]\!] &= \mathbb{T}[\![\tau_1]\!] \to \mathbb{T}[\![\tau_2]\!] \\
\mathbb{T}[\![\exists \vec{\alpha}.\tau]\!] &= \mathbb{T}[\![\tau]\!] \\
\mathbb{T}[\![\tau_1 \times \cdots \times \tau_n]\!] &= \mathbb{T}[\![\tau_1]\!] \times \cdots \times \mathbb{T}[\![\tau_n]\!]
\end{aligned}
$$

Figure 7.3: Existential type translation (removing $\exists$ quantifiers)

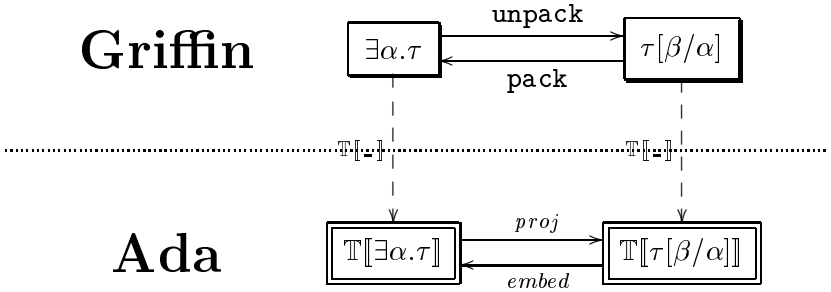


Figure 7.4: Schematic view of the translation of existential types

β, and corresponding abstract base type $\mathcal{B}_\alpha$, we have

$$\mathbb{T}[\![[\beta \mapsto \alpha]\tau]\!] = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau]\!])$$

Proof. By structural induction the proof splits into five cases according to the structure of type expression τ:

$\tau ::= \alpha'$**.** There are two subcases:

- $\alpha' = \alpha$.

  $lhs = \mathbb{T}[\![[\beta \mapsto \alpha]\, \alpha']\!] = \mathbb{T}[\![[\beta \mapsto \alpha]\, \alpha]\!] = \mathbb{T}[\![\beta]\!]$

  $rhs = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\alpha']\!]) = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\alpha]\!]) = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha]\mathcal{B}_\alpha = \mathbb{T}[\![\beta]\!]$

- $\alpha' \neq \alpha$.

$$lhs = \mathbb{T}[\![[\beta \mapsto \alpha]\, \alpha']\!] = \mathbb{T}[\![\alpha']\!] = \mathcal{B}_{\alpha'}$$
$$rhs = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\alpha']\!]) = \mathbb{T}[\![\alpha']\!] = \mathcal{B}_{\alpha'}$$

$\tau ::= K_{\mathbf{Grf}}$.

$$lhs = \mathbb{T}[\![[\beta \mapsto \alpha]\, K_{\text{Grf}}]\!] = \mathbb{T}[\![K_{\text{Grf}}]\!] = K_{\text{Ada}}$$
$$rhs = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha]\, (\mathbb{T}[\![K_{\text{Grf}}]\!]) = \mathbb{T}[\![K_{\text{Grf}}]\!] = K_{\text{Ada}}$$

$\tau ::= \tau_1 \to \tau_2$. By induction hypothesis,

$$\mathbb{T}[\![[\beta \mapsto \alpha]\, \tau_1]\!] = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_1]\!])$$
$$\mathbb{T}[\![[\beta \mapsto \alpha]\, \tau_2]\!] = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_2]\!])$$

so

$$\begin{aligned}
lhs &= \mathbb{T}[\![[\beta \mapsto \alpha](\tau_1 \to \tau_2)]\!] \\
&= \mathbb{T}[\![[\beta \mapsto \alpha]\tau_1 \to [\beta \mapsto \alpha]\tau_2]\!] \\
&= \mathbb{T}[\![[\beta \mapsto \alpha]\tau_1]\!] \to \mathbb{T}[\![[\beta \mapsto \alpha]\tau_2]\!] \\
&= [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_1]\!]) \to [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_2]\!]) \\
&= [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_1]\!] \to \mathbb{T}[\![\tau_2]\!]) \\
&= [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_1 \to \tau_2]\!]) \\
&= rhs
\end{aligned}$$

$\tau ::= \exists\alpha.\tau'$. By induction hypothesis,

$$\mathbb{T}[\![[\beta \mapsto \alpha]\tau']\!] = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau']\!])$$

therefore

$$\begin{aligned} lhs &= \mathbb{T}[\![[\beta \mapsto \alpha](\exists\alpha.\tau')]\!] \\ &= \mathbb{T}[\![\exists\alpha.([\beta \mapsto \alpha]\tau')]\!] \\ &= \mathbb{T}[\![[\beta \mapsto \alpha]\tau']\!] \\ &= [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau']\!]) \\ &= [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\exists\alpha.\tau']\!]) \\ &= rhs \end{aligned}$$

$\tau ::= \tau_1 \times \cdots \times \tau_n$. By induction hypothesis,

$$\mathbb{T}[\![[\beta \mapsto \alpha]\tau_i]\!] = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_i]\!]) \qquad i \in \{1, \ldots, n\}$$

so

$$\begin{aligned} lhs &= \mathbb{T}[\![[\beta \mapsto \alpha](\tau_1 \times \ldots \times \tau_n)]\!] \\ &= \mathbb{T}[\![[\beta \mapsto \alpha]\tau_1 \times \ldots \times [\beta \mapsto \alpha]\tau_n]\!] \\ &= \mathbb{T}[\![[\beta \mapsto \alpha]\tau_1]\!] \times \ldots \times \mathbb{T}[\![[\beta \mapsto \alpha]\tau_n]\!] \\ &= [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_1]\!]) \times \ldots \times [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_n]\!]) \\ &= [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_1]\!] \times \ldots \times \mathbb{T}[\![\tau_n]\!]) \\ &= [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau_1 \times \ldots \times \tau_n]\!]) \\ &= rhs \end{aligned}$$

In all cases of τ we have shown $\mathbb{T}[\![[\beta \mapsto \alpha]\tau]\!] = [(\mathbb{T}[\![\beta]\!]) \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau]\!])$. By the principle of structural induction (a special case of well-founded induction [93]), we conclude that the induction hypothesis holds for all τ.

□

For the packing and unpacking operations in Figure 7.4, we need to find a pair

of coercions between $\mathbb{T}[\![\exists\alpha.\tau]\!]$ $(\mathbb{T}[\![\tau]\!])$ and $\mathbb{T}[\![[\beta \mapsto \alpha]\tau]\!]$ $([\mathbb{T}[\![\beta]\!] \mapsto \mathcal{B}_\alpha](\mathbb{T}[\![\tau]\!]))$. In the following, we will show how to implement this pair of coercion functions by

1. Using the object-oriented features in Ada95 to construct base coercions between a ground type C ($\mathbb{T}[\![\beta]\!]$ in our case) and a base type $\mathcal{B}_\alpha$.

2. Based on these base coercions, inductively construct the coercions between Q and $[C \mapsto \alpha]\,Q$ for an arbitrary target type Q ($\mathbb{T}[\![\tau]\!]$ in our case).

7.2.1 Base coercions through class hierarchy

A brief introduction to the object-oriented programming features of Ada95 is given in Section 6.2.1, which helps to explain the translation algorithm of Griffin existential types.

Superclasses and subclasses are operationally related by *embedding* (embedding of an element of a subclass into a superclass) and *projection* (projection of an element from a superclass to its subclass) as shown in Figure 7.5 [1].

In a λ-calculus with subtyping, the statement $\tau_1 \leq \tau_2$ is traditionally construed as a semantic coercion function of type $[\![\tau_1]\!] \to [\![\tau_2]\!]$ that *projects* (*extracts*) the "τ_2 part" of an element of τ_1. Embedding is also called *upcasting* or *expanding*, while projection called *downcasting* or *narrowing*. With such an operational understanding of subtyping in mind, it is natural to directly use inheritance to implement the embedding and projection pair between an a unique abstract base type and an arbitrary concrete type (implemented as a subclass of the base type).

[1] In an object-oriented paradigm, the embedding operation is usually implicit, and the projection operation appears as an explicit type coercion operation.

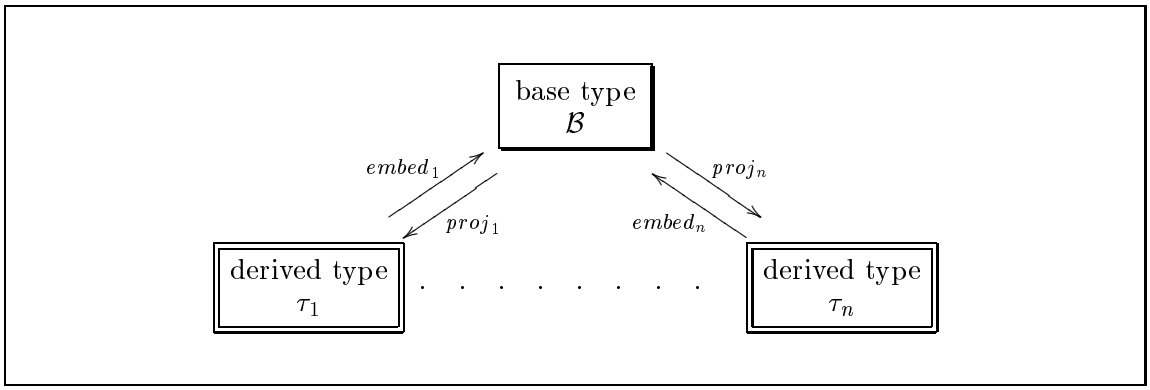


Figure 7.5: Superclass and its subclasses

Projection of an element from a superclass to a subclass is not always safe, but embedding is. The pair can be described as:

$$proj^{\mathcal{B}}_{\tau_i} \circ embed^{\mathcal{B}}_{\tau_i} = id_{\tau_i}$$

$$(embed^{\mathcal{B}}_{\tau_i} \circ proj^{\mathcal{B}}_{\tau_i})(x) = \begin{cases} x & \text{if } x : \tau_i \\ \bot & \text{otherwise} \end{cases}$$

where id_τ is the identity function on τ, $proj^{\tau}_{\tau'}$ is the projection from τ to τ', $embed^{\tau}_{\tau'}$ is the embedding from τ' into τ, $\mathcal{B}$ and τ_i's are the base and derived types respectively.

G2A only translates type-correct Griffin programs, where the typing rules for **pack** and **unpack** ensures that a projection function to a type τ will only be applied to a value which is previously embedded from the same type τ. In the target program, every projection $proj^{\tau}_{\tau'}$ will be performed on a value that was previously formed by the embedding $embed^{\tau}_{\tau'}$. This ensures that the following equation holds in the target program of a G2A translation ($proj \circ embed = id$ always holds):

$$embed \circ proj = id$$

Indeed, if $x = embed(x')$, then

$$\begin{aligned}(embed \circ proj)\, x &= embed(proj(embed(x')) \\ &= embed((proj \circ embed)\, x') \\ &= embed(id(x')) \\ &= x\end{aligned}$$

7.2.2 Coercions for composite types

We present a translation of Griffin existential types based on a *coercion calculus* [41, 40, 94], in which the *primitive coercions*, i.e., *embedding* and *projection*, are implemented using the object-oriented features of Ada95 (see previous section). A *coercion* is the conversion from one type to another. Rather than using the notation $\alpha \to \beta$, we use $\alpha \rightsquigarrow \beta$ to designate the coercion from α to β, to avoid confusion when we apply coercions to function types. An *induced coercion* is a coercion implied inductively by primitive coercions. For instance, if integers are coercible to reals, the boolean-valued functions of real arguments are coercible to boolean-valued functions of integer arguments.

Recall that we only need to find coercions between target type Q (which corresponds to a source existential type) and target type $[C \mapsto \mathcal{B}_\alpha]Q$ (which corresponds to an instance of the existential type) for the translation of Griffin existential types.

Theorem 7.1. *Let Q be a (target) type expression with possible free occurrences of type variables $\mathcal{B}_\alpha$. Given a concrete type C, if we have $proj_C : \mathcal{B}_\alpha \rightsquigarrow C$ and $embed_C : C \rightsquigarrow \mathcal{B}_\alpha$ which are a pair of inverse functions, one can induce a pair of functions $\mathbb{E}_C^Q : [C \mapsto \mathcal{B}_\alpha]Q \rightsquigarrow Q$ and $\mathbb{P}_C^Q : Q \rightsquigarrow [C \mapsto \mathcal{B}_\alpha]Q$ which are inverse of each other.*

Proof. By structural induction the proof splits into four cases according to the structure of type expression Q.

$Q ::= \mathcal{B}_{\alpha'}$. There are two subcases:

- $\alpha' = \alpha$. Define $\mathbb{E}_C^Q = embed_C$ and $\mathbb{P}_C^Q = proj_C$. Since $embed_C$ and $proj_C$ are inverse functions of each other,

$$\mathbb{E}_C^Q \circ \mathbb{P}_C^Q = embed_C \circ proj_C = id_Q$$
$$\mathbb{P}_C^Q \circ \mathbb{E}_C^Q = proj_C \circ embed_C = id_{[C \mapsto \mathcal{B}_\alpha]Q}$$

  $\mathbb{E}_C^Q$ and $\mathbb{P}_C^Q$ are also inverse of each other.

- $\alpha' \neq \alpha$. In this case, $Q = [C \mapsto \mathcal{B}_\alpha]Q = \mathcal{B}_{\alpha'}$. Define $\mathbb{E}_C^Q = \mathbb{P}_C^Q = \lambda x.x : \mathcal{B}_{\alpha'} \to \mathcal{B}_{\alpha'}$, which are apparently a pair of inverse functions.

$Q ::= K$. Define $\mathbb{E}_C^Q = \mathbb{P}_C^Q = \lambda x.x : K \to K = id$. The induction hypothesis is trivially true.

$Q ::= Q_1 \to Q_2$. By the induction hypotheses for type Q_1 and Q_2, there exist $\mathbb{E}_C^{Q_1}$, $\mathbb{E}_C^{Q_2}$, $\mathbb{P}_C^{Q_1}$, and $\mathbb{P}_C^{Q_2}$, where

$$\mathbb{E}_C^{Q_1} \circ \mathbb{P}_C^{Q_1} = id_{Q_1} \quad \text{and} \quad \mathbb{P}_C^{Q_1} \circ \mathbb{E}_C^{Q_1} = id_{[C \mapsto \mathcal{B}_\alpha]Q_1}$$
$$\mathbb{E}_C^{Q_2} \circ \mathbb{P}_C^{Q_2} = id_{Q_2} \quad \text{and} \quad \mathbb{P}_C^{Q_2} \circ \mathbb{E}_C^{Q_2} = id_{[C \mapsto \mathcal{B}_\alpha]Q_2}$$

Define $\mathbb{E}_C^Q f = \mathbb{E}_C^{Q_2} \circ f \circ \mathbb{P}_C^{Q_1}$ and $\mathbb{P}_C^Q g = \mathbb{P}_C^{Q_2} \circ g \circ \mathbb{E}_C^{Q_1}$.

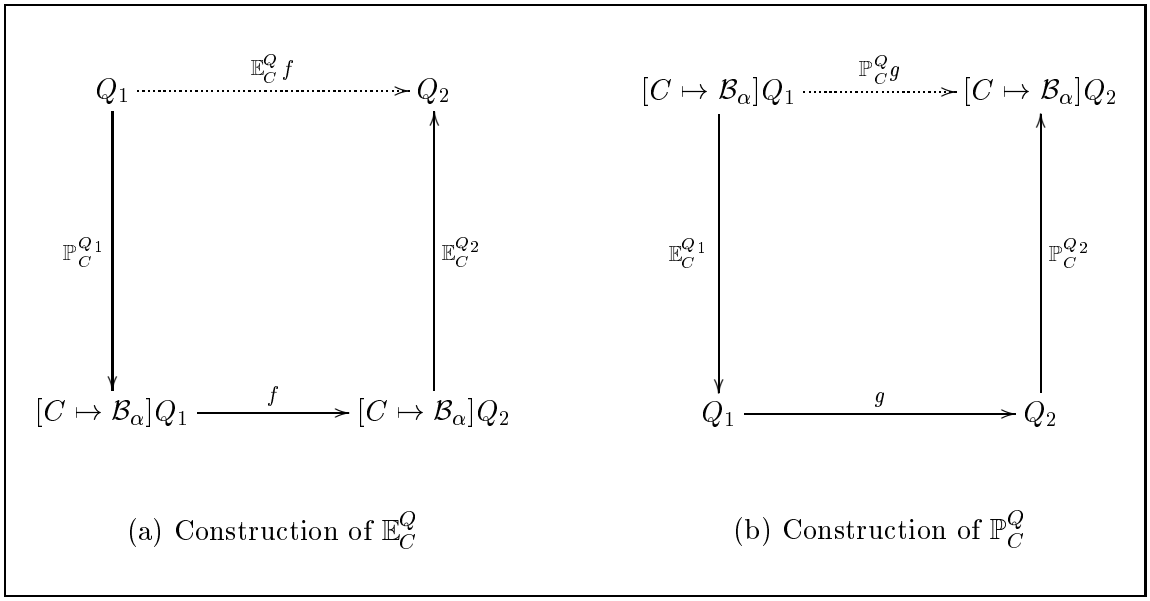
$$\begin{aligned}
(\mathbb{E}_C^Q \circ \mathbb{P}_C^Q) f &= \mathbb{E}_C^Q (\mathbb{P}_C^Q f) \\
&= \mathbb{E}_C^{Q_2} \circ (\mathbb{P}_C^{Q_2} \circ f \circ \mathbb{E}_C^{Q_1}) \circ \mathbb{P}_C^{Q_1} \\
&= (\mathbb{E}_C^{Q_2} \circ \mathbb{P}_C^{Q_2}) \circ f \circ (\mathbb{E}_C^{Q_1} \circ \mathbb{P}_C^{Q_1}) \\
&= id_{Q_2} \circ f \circ id_{Q_1} \\
&= f
\end{aligned}$$

$$
\begin{aligned}
(\mathbb{P}_C^Q \circ \mathbb{E}_C^Q)g &= \mathbb{P}_C^Q(\mathbb{E}_C^Q\, g) \\
&= \mathbb{P}_C^{Q_2} \circ (\mathbb{E}_C^{Q_2} \circ g \circ \mathbb{P}_C^{Q_1}) \circ \mathbb{E}_C^{Q_1} \\
&= (\mathbb{P}_C^{Q_2} \circ \mathbb{E}_C^{Q_2}) \circ g \circ (\mathbb{P}_C^{Q_1} \circ \mathbb{E}_C^{Q_1}) \\
&= id_{[C \mapsto \mathcal{B}_\alpha]Q_2} \circ g \circ id_{[C \mapsto \mathcal{B}_\alpha]Q_1} \\
&= g
\end{aligned}
$$

Therefore,

$$\mathbb{E}_C^Q \circ \mathbb{P}_C^Q = id_Q \qquad \text{and} \qquad \mathbb{P}_C^Q \circ \mathbb{E}_C^Q = id_{[C \mapsto \mathcal{B}_\alpha]Q}$$

Pictorially, this construction of the projection and embedding pair is illustrated in Figure 7.6.



(a) Construction of $\mathbb{E}_C^Q$ (b) Construction of $\mathbb{P}_C^Q$

Figure 7.6: Construction of projection and embedding for function types

This can be expressed directly in Henglein's coercion calculus found in [40] as

$$\frac{c_1 : \alpha \rightsquigarrow \alpha' \qquad c_2 : \beta' \rightsquigarrow \beta}{c_1 \to c_2 \;:\; (\alpha' \to \beta') \rightsquigarrow (\alpha \to \beta)}$$

where c_1 and c_2 are coercions, α, α', β, β' are type expressions. The induced coercion $c_1 \to c_2$ ($\to$ is overloaded to denote coercion construction) is contravariant in the signature of the first coercion argument (negative position).

$Q ::= Q_1 \times \cdots \times Q_n$. By the induction hypotheses for types Q_1 to Q_n, the following equations hold.

$$\mathbb{E}_C^{Q_i} \circ \mathbb{P}_C^{Q_i} = id_Q \qquad \text{and} \qquad \mathbb{P}_C^{Q_i} \circ \mathbb{E}_C^{Q_i} = id_{[C \mapsto \mathcal{B}_\alpha]Q} \qquad i \in \{1, \ldots, n\}$$

Define

$$\mathbb{E}_C^Q(x_1, \ldots, x_n) = (\mathbb{E}_C^{Q_1} x_1, \ldots, \mathbb{E}_C^{Q_n} x_n) \qquad \text{and}$$
$$\mathbb{P}_C^Q(x_1, \ldots, x_n) = (\mathbb{P}_C^{Q_1} x_1, \ldots, \mathbb{P}_C^{Q_n} x_n)$$

Hence

$$\begin{aligned}
(\mathbb{E}_C^Q \circ \mathbb{P}_C^Q)(x_1, \ldots, x_n) &= \mathbb{E}_C^Q(\mathbb{P}_C^Q(x_1, \ldots, x_n)) \\
&= \mathbb{E}_C^Q(\mathbb{P}_C^{Q_1} x_1, \ldots, \mathbb{P}_C^{Q_n} x_n) \\
&= (\mathbb{E}_C^{Q_1}(\mathbb{P}_C^{Q_1} x_1), \ldots, \mathbb{E}_C^{Q_n}(\mathbb{P}_C^{Q_n} x_n)) \\
&= ((\mathbb{E}_C^{Q_1} \circ \mathbb{P}_C^{Q_1})x_1, \ldots, (\mathbb{E}_C^{Q_n} \circ \mathbb{P}_C^{Q_n})x_n) \\
&= (x_1, \ldots, x_n)
\end{aligned}$$

$$\begin{aligned}
(\mathbb{P}_C^Q \circ \mathbb{E}_C^Q)(x_1, \ldots, x_n) &= \mathbb{P}_C^Q(\mathbb{E}_C^Q(x_1, \ldots, x_n)) \\
&= \mathbb{P}_C^Q(\mathbb{E}_C^{Q_1} x_1, \ldots, \mathbb{E}_C^{Q_n} x_n) \\
&= (\mathbb{P}_C^{Q_1}(\mathbb{E}_C^{Q_1} x_1), \ldots, \mathbb{P}_C^{Q_n}(\mathbb{E}_C^{Q_n} x_n)) \\
&= ((\mathbb{P}_C^{Q_1} \circ \mathbb{E}_C^{Q_1})x_1, \ldots, (\mathbb{P}_C^{Q_n} \circ \mathbb{E}_C^{Q_n})x_n) \\
&= (x_1, \ldots, x_n)
\end{aligned}$$

We conclude

$$\mathbb{E}_C^Q \circ \mathbb{P}_C^Q = id_Q \qquad \text{and} \qquad \mathbb{P}_C^Q \circ \mathbb{E}_C^Q = id_{[C \mapsto \mathcal{B}_\alpha]Q}$$

Again, this can also be expressed in the coercion calculus as

$$\frac{c_1 : \alpha_1 \rightsquigarrow \alpha_1{}' \ \ldots \ c_n : \alpha_n \rightsquigarrow \alpha_n{}'}{c_1 \times \ldots \times c_n : (\alpha_1 \times \ldots \times \alpha_n) \rightsquigarrow (\alpha_1{}' \times \ldots \times \alpha_n{}')}$$

where $c_1, \ldots, c_n$ are coercions, $\alpha_1, \ldots, \alpha_n, \alpha_1{}', \ldots, \alpha_n{}'$ are type expressions. The signature of component type extends covariantly to the signature of the induced coercion.

By the principle of structural induction, we conclude that the induction hypothesis holds for all Q.

□

Theorem 7.1 gives a constructive proof for the existence of the induced coercions, i.e., the packing ($\mathbb{E}_C^Q$) and unpacking ($\mathbb{P}_C^Q$) operations in Griffin. They are inverse of each other, i.e., unpacking a previously packed value gives the original value.

7.2.3 An example

The terms *upcasting* and *downcasting* were chosen in the target code in preference to the set-theoretical *projection* and *embedding*, simply because they are more often used in the programming community,

In order to gain better understanding of the the complete Ada target code generated, a more abstract "pseudo target code" is first given in Figure 7.8.

The existential types and the upcasting functions are typeset in a way to make the example more apprehensible. The function $\uparrow$ is an upcasting function with one argument. If the argument is a function f of type τ_2, $\uparrow_{\tau_2}^{\tau_1}(f)$ denotes a function of type τ_1. In other words, $\uparrow_{\tau_2}^{\tau_1}(f)$ returns the appropriate upcast version of its function argument. If the argument is a value of a primitive type, $\uparrow_{\tau_2}^{\tau_1}(x)$ is a base

```
--  negation : bool -> int
fun negation true  => 0
 |  negation false => 1;

--  square : real -> real
fun square (r:real) => r*r;

--  inc : int -> int
fun inc (i: int) => i+1;

var x : ∃ α,β . α × (α → β) × (β → int)
   = pack  α=bool, β=int  in
         α × (α → β) × (β → int)  (false,negation,inc)
     end pack;

var y : ∃ α,β . α × (α → β) × (β → int)
   = pack  α=real, β=real  in
         α × (α → β) × (β → int)  (2.5,square,round)
     end pack;

fun foo p => unpack p as (v,f,f') in  f' (f v)  end unpack;

output(stdOut, foo x);       ----  2
output(stdOut, foo y);       ----  6
```

Figure 7.7: Example of Griffin existential type

coercion. The complete definitions of the above four wrapper functions along with the rest of the translation are fleshed out in Figure 7.9.

7.3 Type translation

The basic translation schemes of the type formers of Griffin, namely *enum*, *alt*, *thread*, *channel*, function types, and *rec* are described in the following sections. When translating Griffin types, auxiliary definitions may have to be generated to conform to the static semantics of Ada.

```
procedure main is
   ----     essentially no change
   function negation(b:boolean) return integer is ... ;
   function square(f:float) return float is ... ;
   function inc(i:integer) return integer is ... ;

   ----     quantified type variables appearing in the existential type
   type α is abstract tagged null record;
   type β is abstract tagged null record;

   ----     instantiate an existential quantified type variable with concrete type τ_α
   --       (* instance of α *)      ---
   type α_concrete is new α with
      record   data : τ_α;   end record;

   function downcast(x:α) return τ_α ... ;
   function upcast(x:τ_α) return α ... ;

   --       (* instance of β *)      ---
   ...

   type existType is
      record
         f1 : α;
         f2 : α  →  β;
         f3 : β  → int;
      end record;

   function foo(p: existType) return integer is
   begin   return p.f3 (p.f2 (p.f1));   end;

   x : existType := (↑^α_bool false, ↑^{α→β}_{bool→int}(negation), ↑^{β→int}_{int→int}(inc));
   y : existType := (↑^α_real 2.5, ↑^{α→β}_{real→real}(square), ↑^{β→int}_{real→int}(round));
begin
   Int_IO.put(foo(x));        ---- output 2
   Int_IO.put(foo(y));        ---- output 6
end main;
```

Figure 7.8: Pseudo code for the translation of the example existential type

7.3.1 Generative and non-generative types

G2A requires access to all Griffin programs before translation starts. Essentially Griffin has structural type equivalence and Ada has name equivalence. All non-

```
with TEXT_IO;
use  TEXT_IO;

procedure main is
   package Int_IO is new Text_IO.Integer_IO(integer);

   ------------     existential type variables     -------------
   generic
      type existTypeVar is abstract tagged private;
   package ExistTypeVarPkg is
      subtype absExistTy is existTypeVar;
      type absExistTyPtr is access all absExistTy'class;
   end ExistTypeVarPkg;

   ------------     alpha     -------------
   type alpha is abstract tagged null record;
   package AlphaPkg is new ExistTypeVarPkg(alpha);

   ------------     beta     -------------
   type beta is abstract tagged null record;
   package BetaPkg is new ExistTypeVarPkg(beta);

   ------------    concrete existential type     -------------
   generic
      with package AbsExistTyPkg is new ExistTypeVarPkg(<>);
      type t is private;
   package ExistConcretePkg is
      type existConcreteType is new AbsExistTyPkg.absExistTy with
         record
            data : t;
         end record;

      function downcast(x: access AbsExistTyPkg.absExistTy'class)
        return t;
      function upcast(x: t) return AbsExistTyPkg.absExistTyPtr;
   end ExistConcretePkg;

   package body ExistConcretePkg is
      function downcast(x: access AbsExistTyPkg.absExistTy'class)
        return t is
      begin
         return existConcreteType(x.all).data;
      end;

      function upcast(x: t) return AbsExistTyPkg.absExistTyPtr is
      begin
         return new existConcreteType'(data => x);
      end;
   end ExistConcretePkg;
```

Figure 7.9: Existential type in Ada95

```
type beta_2_int_ptr is
  access function(x: access BetaPkg.absExistTy'class) return integer;
type alpha_2_beta_ptr is
  access function(x: access AlphaPkg.absExistTy'class)
  return BetaPkg.absExistTyPtr;

------------     existential type     -------------
type existType is
   record
      f1 : AlphaPkg.absExistTyPtr;
      f2 : alpha_2_beta_ptr;
      f3 : beta_2_int_ptr;
   end record;

------------     package instatiations     -------------
package AlphaRealPkg is new ExistConcretePkg(AlphaPkg, float);
package AlphaBoolPkg is new ExistConcretePkg(AlphaPkg, boolean);
package BetaIntPkg   is new ExistConcretePkg(BetaPkg,  integer);
package BetaRealPkg  is new ExistConcretePkg(BetaPkg,  float);

------------     negation : boolean -> integer    -------------
function negation(b: boolean) return integer is
begin
   if b then return 0; else return 1; end if;
end;

function negation_alpha_2_beta(b: access AlphaPkg.absExistTy'class)
                               return BetaPkg.absExistTyPtr is
begin
   return BetaIntPkg.upcast(negation(AlphaBoolPkg.downcast(b)));
end;

------------     square : real -> real    -------------
function square(f:float) return Float is
begin
   return f*f;
end;

function square_alpha_2_beta(r: access AlphaPkg.absExistTy'class)
                             return BetaPkg.absExistTyPtr is
begin
   return BetaRealPkg.upcast(square(AlphaRealPkg.downcast(r)));
end;
```

Figure 7.9: Existential type in Ada95 (continued)

```
    ------------     inc : integer -> integer    -------------
    function inc(i:integer) return integer is
    begin
       return i+1;
    end;

    function incBetaInt(r: access BetaPkg.absExistTy'class)
      return integer is
    begin
       return inc(BetaIntPkg.downcast(r));
    end;

    ------------     round : real -> integer     -------------
    function round(r:float) return integer is
    begin
       return integer(r);
    end;

    function roundBetaReal(r: access BetaPkg.absExistTy'class)
      return integer is
    begin
       return round(BetaRealPkg.downcast(r));
    end;

    ------------     foo : existType    -------------
    function foo(x: existType) return integer is
    begin
       return x.f3(x.f2(x.f1));
    end;

    ------------     x,y     -------------
    x , y : existType;
  begin
    x := (new AlphaRealPkg.existConcreteType'(data => 2.5),
          square_alpha_2_beta'access,
          roundBetaReal'access);
    y := (new AlphaBoolPkg.existConcreteType'(data => false),
          negation_alpha_2_beta'access,
          incBetaInt'access);
    Int_IO.put(foo(x));       ----  2
    Int_IO.put(foo(y));       ----  6
  end main;
```

Figure 7.9: Existential type in Ada95 (continued)

generative Griffin types are first collected. For structurally-equivalent types, a unique type is generated in the target code and all references to those structurally-equivalent types in Griffin program will be replaced by reference to the newly introduced type. For generative types, a unique type in the target Ada program is introduced with the keyword **new**.

7.3.2 Enum types, tuple types and array types

Enumerated types and array types have similar semantics in both languages so the translation is straightforward.

Griffin tuples are fixed length aggregates and can be appropriately represented by Ada records.

7.3.3 Function and procedure types

In Ada95 there are no subprogram types, only access-to-subprogram types. A value of such a type can designate any subprogram matching the profile in the type declaration, whose lifetime does not end before that of the access type. A Griffin function type will be translated into Ada access-to-subprogram type and implicit dereferencing will be done whenever necessary according to the static analysis.

7.3.4 Thread types

If a declaration of a thread type $thread(\tau)$ or a use of it is encountered, a generic Ada package which implements polymorphic threads will first be created; and then instantiated to type τ. The generic package wraps the task which implements the concurrent activity of the thread expression along with the expected return type.

```
generic
   type RetType is private;
package GeneThread is
   type RetTypeFuncPtr is access function return RetType;

   task type thread is
      entry eval(funcPtr: in RetTypeFuncPtr);
      entry value(x:out RetType);
   end thread;
end GeneThread;


package GeneThread is
   task body thread is
      r : RetType;
   begin
      accept eval(funcPtr: in RetTypeFuncPtr) do
         r := funcPtr.all;
      end eval;

      loop
         accept value(x: out RetType) do
            x := r;
         end value;
      end loop;
   end thread;
end GeneThread;
```

Figure 7.10: Thread packages

Whenever a Griffin thread of type τ is seen, a task object whose type is defined in the instantiated package will be created.

The specification and body of the generic package implementing threads are shown in Figure 7.10. The body of the function whose access value is passed to `eval` is the thread expression.

7.3.5 Channel types

Channels are *passive* objects in Griffin so the natural translation to Ada would be a protected type. A protected object is like a *conditional monitor* which provides coordinated access to shared data, through calls on its visible protected operations, which can be protected subprograms or protected entries. A protected unit is declared by a protected declaration which has a corresponding protected body. A protected declaration declares a protected unit, and may be a protected type declaration, in which case it declares a named protected type; alternatively, it may be a single protected declaration, in which case it defines an anonymous protected type, as well as declaring a named protected object of that type [2]. The protected type translation of Griffin channel is:

```
protected type Channel is
   entry Put(e: in ITEM);
   entry Get(e: out ITEM);
   entry Mget(q: out Queue);
private
   ch: Queue;
end Channel;
```

where `Queue` is an instantiation of a generic package implementing queues and `ITEM` is the generic type formal of the generic package in which the above declaration resides.

```
type inttree_enum is (empty,leaf,node);

type inttree;

type acc_inttree is access inttree;

type inttree(tag: inttree_enum) is
   record
      case tag is
         when empty =>
            null;
         when leaf =>
            intElem: integer;
         when node =>
            inttreeElem1 : acc_inttree;
            inttreeElem2 : acc_inttree;
      end case;
   end record;
```

Figure 7.11: Alt type translation

7.3.6 Alt types

Griffin's **alt** type is similar to ML's datatype. To translate the **alt** type we first create an Ada enumeration type whose enumeration literals are the value constructorsof the **alt** type. If the **alt** type being defined is a recursive one, an Ada access type to it will also be created so it can refer to itself in the recursive definition. Finally an Ada discriminated record type containing variant parts will be created which is the translation of the **alt** type. Nullary value constructors are mapped to null records in the variant part and we can distinguish them by the discriminants.

For example, a tree of integers can be algebraically defined as

```
type inttree = alt {empty, leaf(int), node(inttree; inttree)};
```

will be translated into the Ada95 code in Figure 7.11.

For a mutually recursive alt type definition, forward declarations of the access

types of all types being defined are generated before the target code of the **alt** type itself.

7.3.7 Tuple types

Griffin tuple are fixed length aggregates. The translation of the following tuple type

```
type t = (int,real,char);
```

would be:

```
type t is
   record
      t1: integer;
      t2: real;
      t3: character;
   end record;
```

Appropriate projection functions for the translated tuple types should also be generated in the target code.

7.3.8 Other aggregate types

Other aggregate types like *sets*, *bags* and *lists* will be translated into Ada singly linked list of the corresponding element type. For *maps* their Ada counterparts are singly linked lists of a record type with two components whose types are the domain and codomain type of the *map* types.

7.4 Identifiers

In some languages such as Lisp, identifiers are *computable* objects, i.e., one can manipulate identifiers in Lisp programs. Being atoms, Lisp identifiers can be parameters to functions that convert atoms to strings. For example, the following Lisp program yields the value 123

```
(setq abc 123)
(eval (implode '(a b c)))
```

where the built-in function `implode` concatenates a list of atoms. Identifier computation is a kind of reflection, which is always a powerful and dangerous feature (run-time errors can occur because of this kind of reflection). The target code of a language supporting identifier computation needs to carry the textual representation of identifiers at run-time, and run-time environment lookup is also required. In a sense, the mapping of identifiers from the static to dynamic aspects is a "cross-over point" [52] in the semantic equation for identifiers.

However, identifiers are considered to be syntactic entities in most languages (Pascal, C++, ML, Java, Ada, Griffin, etc.) so one cannot compute with identifiers. Usually the valuation function of identifiers simply looks them up in the environment to avoid potential mistakes at compile-time, thus the textual representation of an identifier is neither needed nor available at run-time.

7.5 Parametric overloading

The following function definition of `double` is often used to explain *parametric overloading* (or *bounded polymorphism*):

```
double(x) = x+x
```

ML disallows such function definitions because the type of the operator + is not inferable in the Hindley-Milner type system.

Both Ada95 and Griffin support some sort of bounded polymorphism, which are very different. So there is no straightforward translation of functions with class constraints like the following:

```
fun [mult: A * A -> A] sq(x:A)  =>  mult(x,x)


fun [= : A * A -> bool] eqlist (L1:list[A], L2)  =>
    if head(L1) = head(L2)
    then eqlist(tail L1, tail L2)
    else false
```

Translation of parametrically overloaded functions relies on *arity raising*, which means the original function is translated into a semantically equivalent function with "evidence" parameter (as defined in the discussion of *qualified types* by Mark Jones [47]) corresponding to the constraint passed in.

For brevity, the target code for the two functions above will be depicted in F_ω rather than actual Ada code as shown in Figure 7.12.

When a Griffin polymorphic function definition is encountered, a corresponding Ada95 generic function definition will be created (with the exception of polymorphic functions involving row variables of extensible records, which will be addressed in Section 7.7). When a function call to that function is first seen, the generic function

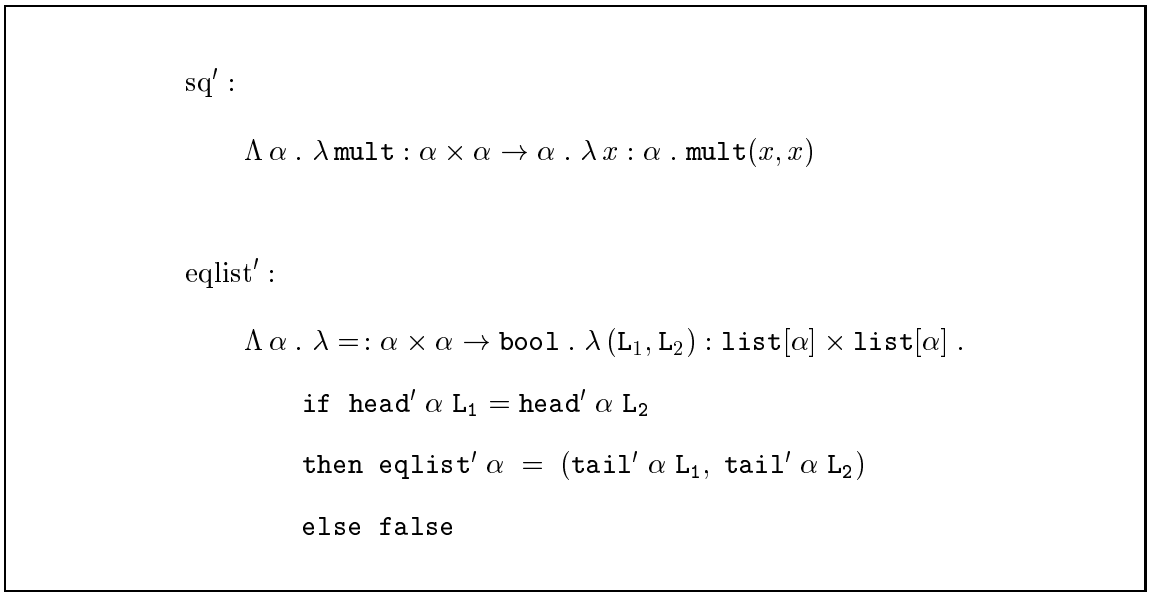


Figure 7.12: Arity raising

will be instantiated to the appropriate type and a corresponding function call to the instantiated function will be generated. Further function calls with the same parameter types are simply translated into calls to that function.

The class constraints on the Griffin side are checked by the semantic analyzer and functions are arity raised in the target programs. There will no be run-time errors resulting from the violation of the class constraint, because the existence of certain functions is guaranteed by the success of the static semantics check.

7.6 Closure conversion

Griffin functions are more than code; they are composed of an environment as well as code. A *closure* formalizes the notion of freezing the values of the free variables in function bodies; whenever the value of a free variable is needed, it will be taken from the saved environment.

Closure conversion (or *lambda lifting*, *functional defunctionalization*) is the process of converting a function that refers to non-local variables into a function without such references. Without going into all of the pragmatic issues associated with different approaches to closure conversion, we first investigate the general form of it. In λ-calculus, *combinator* is a λ-expression in which there are no occurrences of free variables. λ-lifting "lifts out" (abstracts) the free variables from possibly nested λ-abstraction to form a new combinator. Let $fv(E)$ denote the set of free variables in the expression E. The "lifting" function, $\mathcal{L}$, therefore has the following definition:

$$
\begin{array}{rcll}
\mathcal{L}(E) & = & E & \text{if } E \text{ is in applicative form} \\
\mathcal{L}(\lambda x_1 \ldots x_m.E) & = & \alpha\ v_1 \ldots v_n & \text{if } E \text{ is an applicative expression} \\
 & & & \text{and } fv(E) - \{x_1, \ldots, x_m\} = \\
 & & & \{v_1, \ldots, v_n\} \\
 & & & \text{where } \alpha\ v_1 \ldots v_n\ x_1 \ldots x_m\ = E \\
\mathcal{L}(\lambda x_1 \ldots x_m.E) & = & \mathcal{L}(\lambda x_1 \ldots x_m.\mathcal{L}(E)) & \text{if } E \text{ is neither an} \\
 & & & \text{applicative expression} \\
 & & & \text{nor a } \lambda\text{-abstraction} \\
\mathcal{L}(E_1 E_2) & = & \mathcal{L}(E_1)\mathcal{L}(E_2) & \text{for } \lambda\text{-expressions } E_1, E_2 \\
\mathcal{L}((E)) & = & (\mathcal{L}(E)) & \text{for } \lambda\text{-expression } E
\end{array}
$$

Ada95 does not support closures so closure conversion must be performed in our translation. We have to deal with the indefinite extent of functions and their free variables. In order to close an open function, it is represented by a closure, a data structure containing both the pointer to its code and a record for its free variables. In other words, the dependency on the environment is made explicit. Since the lifetimes of the closures are not nested, all variables should be flatly heap allocated.

A straightforward approach would convert closures into records which contain a

pointer to the function body, the environment and the original function parameters. However, we would also like the closure-converted program to be well-typed according to the rules of the source language — rules that should also be enforceable in the target language. The difficulty is that two functions with the same type might well differ in the number and types of their free variables, and hence have closure records of completely different structural types in the target code. The following simple example helps explain the problems involved:

$$\begin{aligned} f_1 &= \lambda x{:}int\ .\ \lambda y{:}int\ .\ x + z \\ f_2 &= \lambda x{:}int\ .\ \lambda y{:}int\ .\ 1 \end{aligned}$$

z is of type int, the types of f_1 and f_2 are both $int \to int \to int$. However, if we adopt the intuitive approach to convert closures to records containing pointers to its free variables, f_1 and f_2 would have different types ($f_1 : int \to int \to int \to int$, $f_2 : int \to int \to int$) in the target program since f_1 has a non-local reference and f_2 has none.

Minamide, Morrisett, and Harper [59] have treated this problem, but their solutions rely either on new language primitives for closure manipulation, which complicate subsequent optimization, or on giving closure existential types, adding another dimension of complexity to the type system. Worse yet, neither solution leads to typable Ada.

Let g_1 and g_2 further denote the function bodies of f_1 and f_2 respectively.

$$\begin{aligned} g_1 &= \lambda y{:}int\ .\ x + z \\ g_2 &= \lambda y{:}int\ .\ 1 \end{aligned}$$

Another possible approach that we have considered but rejected is the use of *closure algebraic datatypes* proposed by Bell, Bellegarde and Hook [12]. Their approach

relies on having the whole program available for analysis and translation. The basic idea is to represent function closures as members of a *closure algebraic data type* (i.e., discriminated union). There is one constructor corresponding to each function definition in the original program; its arguments are the free variables in the function body. All higher-order operations on functions are replaced by equivalent operations on closure values. Function definitions are lambda-lifted, with additional parameters of closure constructor applications, which are the wrapping of the free variables.

The translation of the functions f_1 and f_2 mentioned above and their applications is shown in Figure 7.13.

```
datatype closure_g =  G1 of {z:int, x:int}
                    | G2 of {}
and      closure_f =  F1 of {z:int}
                    | F2 of {}

fun g1(z,x,y) = x+z
fun g2(y)     = 1
fun f1(z,x) = G1 {z=z, x=x}
fun f2(x)   = G2 {}

fun apply_g(G1 {z,x}, y) = g1(z,x,y)
 |  apply_g(G2 {}, y)    = g2(y)

fun apply_f(F1{z}, x) = f1(z,x)
 |  apply_f(F2{} , x) = f2(x)
```

Figure 7.13: Closure algebraic datatypes

A type-preserving translation for closures is not as easy as it sounds. A naive translation from closures to records would violate the typable target code property (Section 3.4). We choose a different approach to closure conversion which enables us to drop the whole-program transformation requirement and at the same time,

achieves the goal of typable Ada95 code. The idea is to take advantage of the object-oriented features of Ada95: Griffin functions of the same type are assigned a type corresponding to a unique abstract base type in the target code. Each concrete derived type of the abstract base type extends the base type with a field for each free variable. Each function invocation `foo(bar)` will be translated into the form `apply(foo',bar)`, wherein `foo'` is the closure representation of `foo`.

As a running example, consider the Griffin functions `f1` and `f2` shown in Figure 7.15. Function `f1` has a free variable whereas `f2` has none. Dynamic dispatching in Ada95 is achieved through the use of the "class" mechanism and pointers. Functions of type `int -> int` are translated into pointers to objects of abstract type `Closure_F` in the target program as

```
type Closure_F is abstract tagged null record;
type Closure_F_ptr is access all Closure_F'class;
function apply(c:Closure_F, x:integer) return integer is abstract;
```

where the function `apply` (associated with type `Closure_F`) dynamically dispatches. For dynamic dispatching purposes, we have to define the pointer to `Closure_F` type, `Closure_F_ptr`. The class relation among those "closure types" is shown in Figure 7.14.

The type `Closure_F1` represents Griffin functions of type `int -> int` with a free variable of type `int` is defined by extending the type `Closure_F` as follows:

```
type Closure_F1 is new Closure_F with
   record
      fv1 : integer;
```

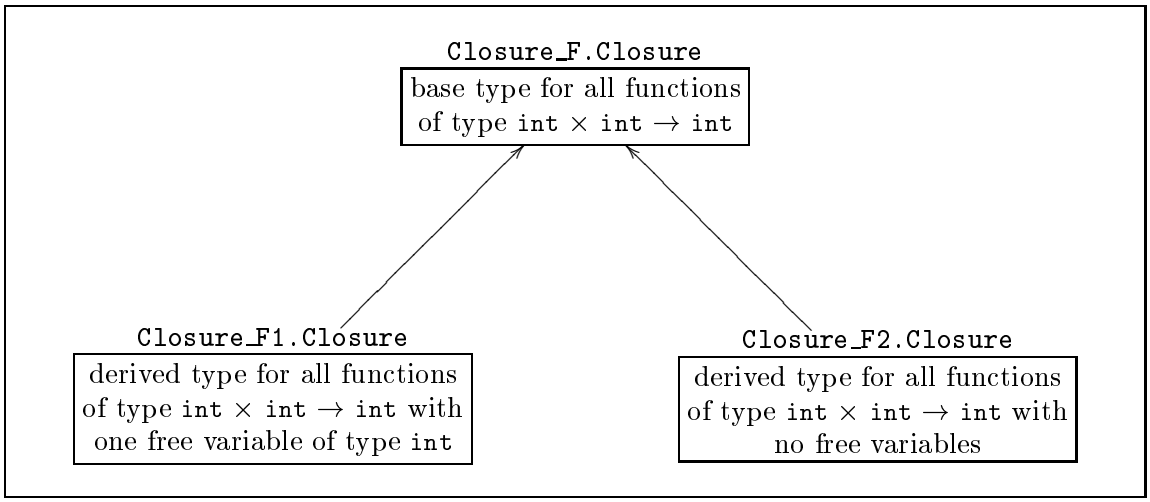


Figure 7.14: Class relation among closure types

```
end record;
```

For symmetry, we also define the type `Closure_F2` (essentially the same type as `Closure_F`) for Griffin functions of type `int -> int` with no free variables (also by extending `Closure_F`, but with a null record):

```
type Closure_F2 is new Closure_F with null record;
```

Both `Closure_F1` and `Closure_F2` are derived types of `Closure_F`. Now we can define `f1` and `f2` as

```
f1,f2 : Closure_F_ptr;
```

and obtain a typable Ada95 target program because `f1` and `f2` have the same type. Evidently, this approach can only be applied to target languages supporting the object-oriented paradigm. As an example of the translation of closure conversion, the target code for the following Griffin program is shown in Figure 7.15.

```
procedure closureConversion is
   generic
      type Domain is private;
      type Codomain is private;
   package GeneClosure is
      type Closure is abstract tagged null record;
      type ClosurePtr is access all Closure'class;
      function apply(c:Closure; d:Domain) return Codomain is abstract;
   end GeneClosure;

   package Closure_F_Pkg is new GeneClosure(integer,integer);

   package F1_Pkg is
      type Closure_F1 is new Closure_F_Pkg.Closure with
         record  z : integer;  end record;
      function apply(c:Closure_F1; x:integer) return integer;
   end F1_Pkg;

   package body F1_Pkg is
      function apply(c:Closure_F1; x:integer) return integer is
         begin  return x+c.z;  end;
   end F1_Pkg;

   package F2_Pkg is
      type Closure_F2 is new Closure_F_Pkg.Closure with null record;
      function apply(c:Closure_F2; x:integer) return integer;
   end F2_Pkg;

   package body F2_Pkg is
      function apply(c:Closure_F2; x:integer) return integer is
         begin  return 1;  end;
   end F2_Pkg;

   f1,f2 : Closure_F_Pkg.ClosurePtr;
   z : integer := 2;
   use Closure_F_Pkg;
begin
   f1 := new F1_Pkg.Closure_F1'(z=>z);
   f2 := new F2_Pkg.Closure_F2;
   Int_IO.put(apply(f1.all,3));          ----  5
   Int_IO.put(apply(f2.all,3));          ----  1
end closureConversion;
```

Figure 7.15: Object-oriented approach for closure conversion

```
var z: int := 2;

fun f1 (x:int) => x+z;

fun f2 (x:int) => 1;
```

```
f1(3);                    ----  5
f2(3);                    ----  1
```

7.7 Extensible records

Just as in Haskell or ML, it is not clear how to derive extensible records in the Ada type system without any serious change. Extensible records are useful in certain situations. For example, a common way to simulate many of the effects of inheritance in ML would be greatly simplified if extensible records were supported in ML. If `superClass` has the following definition in ML:

```
type superClass =
   {
     l1 : τ1
     ⋮
     ln : τn
   }
```

and `subClass` is derived from `superClass` so that `subClass` inherits all the fields of `superClass`, and extends it with one field l_{n+1} as follows:

```
type subClass =
   {
      l1   : τ1
      ⋮
      ln   : τn
      ln+1 : τn+1
   }
```

The natural way to extend `superClass` with a new field is to write a function which verbosely list all the fields as the following:

```
fun createSubClass (sup:superClass) vn+1 =
```

```
({
   l1   = #l1 sup
   ⋮
   ln   = #ln sup
   ln+1 = vn+1
}) : subClass
```

However, if ML supports extensible records, the above function can be significantly simplified as:

```
fun createSubClass (sup:superClass) vn =
    sup\ln+1 with {ln+1 = vn+1}
```

where $r \setminus l$ removes the field l from record r if r contains field l; otherwise it returns r.

However, if the source language supports inheritance then the above argument for the usefulness of row-variable polymorphism does not hold. In Ada83, since records can be manipulated as whole objects and Ada has copy semantics for assignments, the above problem can be easily addressed by a record assignment statement.

It is not obvious how to translate a Griffin polymorphic function involving row variables to an Ada generic function. The generic discriminated record type is not powerful enough to simulate extensible records because it limits one to components of discrete types; neither is the generic formal type parameter which can be an arbitrary extension of a tagged type in Ada95 as follows:

```
type parent is tagged record
   l1 : τ1
   ⋮
   ln : τn
end record;
```

```
generic
   --  unspecified extension
   type T is new parent with private;
package GP is
   ⋮
end GP;
```

The reason that this scheme will not work is that all record types are required to be an extension of some tagged type. The translation of record with only one component is obvious and its Ada counterpart is simply a tagged type with the same component. For a record type with more than one component multiple inheritance must be employed. This method of translation requires multiple inheritance to work. Unfortunately multiple inheritance is not supported in Ada95 and simulating the effect of it is asymmetric [2]. Ideally, Griffin functions would be translated into Ada95 functions, with polymorphic functions being translated into generic functions. In G2A polymorphic functions definitions involving row variables are translated using different approaches. The correctness of those function definitions are checked in the static analysis. However, rather than translating Griffin polymorphic functions involving row-variables into Ada generic functions, G2A create monomorphic instances for all uses(invocations as well as passing around).

In the following code, there is a row-variable polymorphic function `foo`:

```
fun foo r => #l r;

var r1 : {i:int; l:real; b:bool} := {i=1, l=2.0, b=true};
var r2 : {i:int; l:char; b:bool} := {i=1, l='a', b=false};
var r3 : {l:int; c:real}         := {l=4, c=5.0};

foo r1;       --  2.0
foo r2;       --  'a'
foo r3;       --  4
```

The type of `foo` is

$$\forall \alpha, \rho \,.\, \rho \backslash \mathtt{l} \Rightarrow \{\mathtt{l} : \alpha;\ \rho\} \rightarrow \alpha$$

where $\rho\backslash\mathtt{l}$ denotes a type constraint as defined by Mark Jones [47], which specifies that the row denoted by ρ does not have a field labelled `l`.

We present two possible non-type-preserving translations for record polymorphism. The first one shown in Figure 7.16 is type-safe. However, if there exist many uses of a row-variable polymorhic function, the code generated becomes pretty lenghty due to the monomorphic instances required. The other one given in Figure 7.17 is type-unsafe, which employs unchecked conversion but generates a shorter target code by instantiating instances of a geneic function rather than having many copies of almost identical monomorphic functions.

The type-safe translation is straightforward by looking at the target code generated. In the type-unsafe translation, all Griffin records are translated into Ada linked lists, thus record field selection is implemented as list element projection. Griffin is a statically-typed language, therefore the type of the actual parameter of `foo` is statically decidable. All the Griffin record types will be pre-processed in G2A so they conform to a canonical form: all the fields in a record are listed in their alphabetical order. The target function generated for `foo` will be arity-raised and the extra parameter denotes the "offset" of the record element (starting from 0). Records are translated into linked list of pointers rather than tuples for two reasons: records are heterogeneous aggregates, and tuples are fixed length aggregates. The nodes in the list contain a field `data` which is a pointer to the actual record field. Since record fields may have different types, unchecked conversions have to be performed so all the elements in the list are of the same type.

```
with Text_Io;    use Text_Io;

procedure main is
   package Int_Io is new Integer_IO (Integer);   use Int_Io;
   package Flt_Io is new Float_IO (Float);       use Flt_Io;

   type recType1 is
      record
         b: boolean;
         i: integer;
         l: float;
      end record;

   type recType2 is
      record
         b: boolean;
         i: integer;
         l: character;
      end record;

   type recType3 is
      record
         c: float;
         l: integer;
      end record;

   r1: recType1 := (true,  1, 2.0);
   r2: recType2 := (false, 1, 'a');
   r3: recType3 := (5.0,   4     );

   function foo(r: recType1) return float is
   begin
      return r.l;
   end;

   function foo(r: recType2) return character is
   begin
      return r.l;
   end;

   function foo(r: recType3) return integer is
   begin
      return r.l;
   end;

begin
   put(foo(r1));        -- 2.0
   put(foo(r2));        -- 'a'
   put(foo(r3));        -- 4
end Main;
```

Figure 7.16: Type-safe translation of Griffin record polymorphism

```
⋮
generic
   type FieldType is private;
function GeneFoo(r: RecList; offset:integer) return FieldType;

function GeneFoo (r: RecList; offset:integer) return FieldType is
   i : integer := 0;
   curNode : RecList;
begin
   while (i /= offset)
      curNode := curNode.next;
   end loop;

   return curNode.data
end GeneFoo;

--  creat a monomorphic instance
function foo is new GeneFoo(float);

--  Record fields are sorted alphabetically.
--  Create pointers ptr1, ptr2, ptr0 to integer 1, real 2.0,
--  and boolean true respectively.
--  Perform unchecked conversions so all pointers have the
--  same type.
r1 : RecList :=
   insert(ptr0, insert(prt1, insert(ptr2, emptyList)));

--  Create pointers ptr1, ptr2, ptr0 to integer 1, character 'a',
--  and boolean false
r2 : RecList :=
   insert(ptr0, insert(prt1, insert(ptr2, emptyList)));

--  Create pointers ptr1, ptr0 to integer 4, and real 5.0
r3 : RecList :=
   insert(ptr0, insert(prt1, emptyList)));

--  offset starts from 0
put(foo(r1,2).all);        -- 2.0
put(foo(r2,2).all);        -- 'a'
put(foo(r3,1).all);        -- 4
```

Figure 7.17: Pseudo code for the type-unsafe translation of Griffin record polymorphism

7.8 Channels

Griffin channels are described in detail in Section 5.1.6. Channels are first-class values, which provide bi-directional anonymous communication between multiple readers and multiple writers.

7.8.1 Synchronous channels

Protected objects of Ada95 (similar to conditional monitors) support concurrent programming. With protected objects, the unsatifactory polling and race conditions often found in parallel activities are eliminated. The syntax of protected objects is similar to that of a package or a task. This consists of a specification describing the interface, and a body depicting the dynamic behaviour of the protected object. Besides the interface, the private part of the specification of a protected object or a protected type contains the hidden shared data and the specification of subprograms and entries which can only be used in the object. The main distinction between a procedure and a function in the protected body is that a procedure can access the private data in an arbitrary manner whereas a function is only allowed to read the private data [2].

The syntax of an entry body is similar to that of a procedure body except that it always has a barrier consisting of the keyword **when** followed by a boolean expression. At the end of the execution of a procedure body or an entry body of a protected object, all barriers which have queued tasks are re-evaluated thus permitting the processing of an entry call which had been queued on a false barrier.

The dynamic behaviour of protected objects can be understood as a two-level eggshell model [9]. We can picture that a protected object with its entry queues are

surrounded by a shell, and the shell can only be penetrated by a new task trying to call a subprogram or entry when the protected object is quiescent. Tasks can thus be waiting at two levels, outside the shell where they are just milling around in an unstructured way contending for access to the implementation lock which guards the protected object as a whole, and inside the shell in an orderly manner on entry queues. The internal waiting tasks always take priority over the external tasks.

In G2A, a Griffin channel is implemented as a passive protected object rather than a hidden thread and its associated buffer; therefore the impact on scheduling is clear from its specification. The channel buffer is implemented as a FIFO queue. We use the **requeue** statement (new addition to Ada95) to program so-called "preference control" [9] for modelling the semantics of operations **put**, **get**, and **multiple get** on Griffin synchronous channels as shown in Figure 7.18.

If a Griffin thread performs a **get** or **multiple get** on a synchronous channel, the barrier condition is that the channel is not empty. However, for **multiple get**, the thread will be requeued so it can unblock threads that had been previously queued on the **PutWait** entry.

For a thread that **put**s datum into a channel, it is requeued on the **PutWait** entry automatically. Intuitively, the two boolean conditions **getDone** and **mGetDone** will be set to true only after a thread performs **get** or **multiple get** respectively, which in turn release the thread that is blocked after executing a **put**.

7.8.2 Asynchronous channels

The translation scheme for asynchronous channels is simpler than that for the synchronous ones and no agents are required:

```
generic
   type Item is private;
package GeneSyncChannel is
   Package QueuePkg is new GeneQueue(Item);
   use QueuePkg;

   protected SyncCh is
      entry Put(i: in Item);
      entry Get(i: out Item);
      entry Mget(q : out Queue);
   private
      entry PutWait;
      entry MgetWait;
      ch       : Queue;
      getDone  : boolean := false;
      mGetDone : boolean := false;
   end SyncCh;
end GeneSyncChannel;

package body GeneSyncChannel is
   protected body SyncCh is
      entry Put(i: in Item) when true is
      begin
         QueuePkg.insert(ch,i);
         requeue PutWait;
      end Put;

      entry PutWait when (getDone or mGetDone) is
      begin
         getDone := False;
      end;

      entry Get(i: out Item) when not(isEmpty(Ch)) is
      begin
         QueuePkg.remove(ch,i);
         getDone := true;
      end;

      entry Mget(q: out Queue) when not(IsEmpty(Ch)) is
      begin
         QueuePkg.clone(ch,q);
         QueuePkg.init(ch);
         mGetDone := true;
         requeue MgetWait;
      end;

      entry MGetWait when PutWait'Count = 0 is
      begin
         mGetDone := false;
      end;
   end SyncCh;
end GeneSyncChannel;
```

Figure 7.18: Translation of Griffin synchronous channels

- When a Griffin asynchronous channel declaration is encountered, an Ada95 generic asynchronous channel package will be instantiated to appropriate types.
- For each asynchronous channel **send**, a call to the channel manager's **put** entry is created. The guard condition is `true`, so that the evaluation of the barrier is always true and therefore the sender will not be blocked.
- Each Griffin asynchronous channel **recv** is translated into an entry call to the channel manager's **get** entry. The barrier condition blocks the receiver if the channel buffer is empty or returns an element if not.
- The case of **multiple recv** is very similar to that of **recv** except if there is more than one item in the queue then all the elements in the channel buffer will be returned.

7.9 Garbage collection

G2A does not deal with garbage collection for the time being because an Ada95 implementation may have the garbage collector built in. Otherwise there already exists commercial products translating Ada95 programs to Java byte code, in this way garbage collection is taken care of by the Java virtual machine. If it really comes to the point that we should deal with the garbage collection problem in G2A, then all memory management routines could be done via Boehm's conservative garbage collector [15] (by the use of C interface pragmas in the generated Ada code).

Some straightforward compile-time garbage collection may be performed for the allocation of maps, sets and bags in the translation program.

```
--   file :: asyncchannel.ads

with GeneQueue;
with Text_IO;

generic
   type ITEM is private;
package AsyncChannel is
   package ItemQueue is new GeneQueue(Item);
   use ItemQueue;

   protected Channel is
      entry Put (e: in ITEM);
      entry Get (e: out ITEM);
      entry Mget(q: out Queue);
   private
      ch: Queue;
   end Channel;
end AsyncChannel;
```

```
--   file :: asyncchannel.adb

with GeneQueue;
with Text_IO;

package body AsyncChannel is
   protected body Channel is
      entry Put(e: in ITEM) when true is
      begin
         insert(ch,e);
      end;

      entry Get(e: out ITEM) when not isEmpty(ch) is
      begin
         remove(ch,e);
      end;

      entry Mget(q: out Queue) when not isEmpty(ch) is
      begin
         copy(ch,q);
         init(ch);
      end;
   end Channel;
end AsyncChannel;
```

Figure 7.19: Translation of Griffin asynchronous channels

7.10 Exceptions

Griffin provides a type-safe, dynamically scoped exception mechanism which allows programs to handle unusual or deviant conditions. Griffin exceptions (described in Section 5.1.8) are first-class values. To translate Griffin exceptions we have to associate with each Griffin thread (Section 7.8) a global stack, which retains the values for successively raised exceptions.

7.11 Pattern matching

Consider the familiar ML function `length` yielding the number of elements in a homogeneous list:

```
fun length [] = 0
  | length (x::xs) = 1 + (length xs)
```

The function argument is deconstructed to determine which branch should be evaluated. If the argument is an empty list, integer zero is returned; otherwise the function returns the summation of one and the length of the "tail" of the argument.

A pattern serves two purposes: firstly it specifies the form that a formal argument must take before the corresponding branch can be applied; secondly it has the effect of deconstructing the argument and introducing new bindings (except when the pattern is a nullary data constructor or a literal).

In Griffin, the syntax for pattern-matching formal parameter in a function definition is as follows:

```
fun foo s =>
    case s of
        P1 | C1 => E1;          ||
        ⋮
        Pn | Cn => En;
```

wherein each $P_i \,|\, C_i$ `=>` E_i is a pattern branch, P_i is a pattern, C_i is a boolean-valued expression and $P_i \,|\, C_i$ is called a guarded pattern. The expression `s` that is being matched against is called the "subject". In Griffin, pattern matching follows the order in which the pattern branches are written down in the source code. When pattern-matched against each pattern branch, the subject is deconstructed according to the pattern P_i. If the deconstruction succeeds, then the boolean condition C_i is tested with possibly newly introduced bindings resulting from the pattern deconstruction. Only if it is true, the pattern branch expression E_i is evaluated and the result is the value of the entire pattern matching expression. If none of the pattern branches matches, an exception is raised.

In order to make the presentation of the semantic equations for pattern-matching terse and more readable, the following notations are used:

$\langle \ldots \rangle$ **(Syntax Construction)** It encloses a piece of unevaluated syntax, which can contain occurrences of computations $\ulcorner \ldots \lrcorner$ (see definition below). This is similar to the Scheme back quote "`" operator.

$\ulcorner \ldots \lrcorner$ **(Escape in Syntax Construction)** Used inside a syntax construction. It encloses a computation that will be filled into its place in the syntax construction. This is similar to the Scheme anti-quote "," operator.

$\rho\,[x \mapsto \tau]$ **(Environment Perturbation)** The environment is a function mapping identifiers to its associated values. The notation, $\rho\,[x \mapsto \tau]$, denotes the function ρ perturbed to map x to τ. In most block-strucutured programming languages, environment perturbation is environment enriching.

$\boxed{\mathrm{x}}$ **(Unique identifier)** All occurrences of $\boxed{\mathrm{x}}$ within a semantic equation denote an identifier, which is different from all other identifiers.

$+\!\!+$ **(Overloaded Aggregate Concatenation)** We use $+\!\!+$ to denote the overloaded concatenation operator on lists, tuples, and environments. List concatenation is common in many languages and thus need not be elaborated any further.

If τ_1 is the tuple type $\tau_{11} \times \ldots \times \tau_{1m}$, and τ_2 is another tuple type $\tau_{21} \times \ldots \times \tau_{2n}$, then the tuple concatenation operator $+\!\!+$ has the following type:

$$+\!\!+ \quad : \quad (\tau_{11} \times \ldots \times \tau_{1m}) \rightarrow (\tau_{21} \times \ldots \times \tau_{2n}) \rightarrow (\tau_{11} \times \ldots \times \tau_{1m} \times \tau_{21} \times \ldots \times \tau_{2n})$$

Sometimes we use $(\ldots)_n$ to explicitly annotate that the length of a tuple is n.

The lookup in an environment ρ is a partial function mapping an identifier to its associated value. If x is defined in environment ρ (or x is in the domain of ρ), we say $x \in \rho$. The result of environment concatenation (or environment composition) $\rho_1 +\!\!+ \rho_2$ is another environment, in which a lookup will be first performed in ρ_1, if an associated value is found, then return that value. Otherwise, search ρ_2 and the result will be the result for the entire lookup. The

lookup in a concatenated environment can thus be defined as

$$(\rho_1 +\!\!+ \rho_2)(x) = \begin{cases} \rho_1(x) & \text{if } x \in \rho_1 \\ \rho_2(x) & \text{otherwise} \end{cases}$$

Furthermore, we define $\tilde{+\!\!+}$ to be a binary aggregate concatenation operator which requires its arguments to be disjoint. That is, $A_1 \tilde{+\!\!+} A_2$ requires that $A_1 \cap A_2 = \emptyset$.

π_i **(Tuple Projection)** If t is a tuple of length n, $\pi_i(t)$ where $1 \leq i \leq n$ yields the i^{th} component of tuple t.

$+\!\!+\!\!+$ **(Pairwise Concatenation of Two Aggregates)** With SETL notation, the pairwise concatenation operator on sets can be straightforwardly defined as $\mathrm{X} +\!\!+\!\!+ \mathrm{Y} = \{\mathrm{a} +\!\!+ \mathrm{b} \mid \mathrm{a} \in \mathrm{X}, \mathrm{b} \in \mathrm{Y}\}$, where $+\!\!+$ is the overloaded concatenation operator defined above. For example, if $\forall i \in 1 \ldots n, A_i = (a_{i1}, a_{i2}, \ldots, a_{in})$, then

$$A_1 +\!\!+\!\!+ A_2 +\!\!+\!\!+ \ldots A_m = [(a_{11}, \ldots, a_{m1}), (a_{12}, \ldots, a_{m2}), \ldots, (a_{n1}, \ldots, a_{nm})]$$

It can be defined in ML as follows:

```
fun conc l = foldl (fn (l1,l2) => l1 @ l2) [] l;


fun +++ list1 list2 =
    conc (map (fn s => map (fn t => s @ t) list2) list1)
```

where `@` is the list concatenation operator in ML.

Shown below are the semantic equations for function body ($\mathbb{FB}$) of the above form, pattern matching ($\mathbb{PM}$), pattern branch ($\mathbb{PB}$), and pattern ($\mathbb{P}$).

$\mathbb{FB}$: funBodyAST $\rightarrow$ env $\rightarrow$ (mty,IACTION)

$\mathbb{FB}$ $[\![$ **case** s **of** P_1 "|" C_1 "=>" E_1 "||"..."||" P_n "|" C_n "=>" E_n $]\!]$ ρ =
  let
    (sMty,sIA) = $\mathbb{E}$ $[\![$s$]\!]$ ρ
  in
    $\mathbb{PM}$ $[\![$$P_1$"|"$C_1$"=>"$E_1$ "||"..."||" P_n"|"C_n"=>"$E_n$$]\!]$ (sIA,sMty) ρ
  end;

$\mathbb{PM}$: patBrAST list $\rightarrow$ (IACTION $\times$ semMty) $\rightarrow$ env $\rightarrow$ (mty,IACTION)

$\mathbb{PM}$ $[\![$patBrs$]\!]$ (sIA,sMty) ρ =
  let
    (pbsMty,caseBrsIA) = $\mathbb{PB}s$ $[\![$patBrs$]\!]$ (x,sMty) ρ
  in
    (
      pbsMty,
      ⟨
        Let
          x = ⌜sIA⌟
        In
          ⌜caseBrsIA⌟
        End
      ⟩
)
  end;

$\mathbb{PB}s$: patBrAST list $\rightarrow$ (Name,semMty) $\rightarrow$ env $\rightarrow$ (mty,IACTION)

$\mathbb{PB}s$ $[\![$ [ ] $]\!]$ (sName,sMty) ρ = ⟨Raise PatExhaustExn⟩

$\mathbb{PB}s$ $[\![$patBr "||" patBrs$]\!]$ (sName,sMty) ρ =
  let
    (pbMty,caseBrIA) = $\mathbb{PB}$ $[\![$patBr$]\!]$ (sName,sMty) ρ
    (pbsMtys,caseBrsIA) = $\mathbb{PB}s$ $[\![$patBrs$]\!]$ (sName,sMty) ρ
  in
    if typeConformant(pbMty::pbsMtys) then
      (
        pbMty,
        ⟨
          Case ⌜caseBrIA⌟ Of
            None => ⌜caseBrsIA⌟

```
                SOME [x] => [x]
             End Case
          ⟩
       )
    else
       errorMsg("Case branches type mismatch")
end;
```

ℙℬ : patBrAST → (Name,semMty) → env → (mty,IACTION)

```
ℙℬ ⟦pat "|" guardCond "=>" patBrBody⟧ s ρ =
   let
      (pbMty,valuesIA,boundVarNames,ρ') = ℙ ⟦pat⟧ s ρ
      ((_,BoolType), guardCondIA) = 𝔼 ⟦guardCond⟧ (ρ + ρ')
      (patBrBodyMty,patBrBodyIA) = 𝔼 ⟦patBrBody⟧ (ρ + ρ')
   in
      (pbMty,
       ⟨
         Let
            [x] ← ⌜valuesIA⌟
            [r] ← NONE
         In
            For [e] In [x] Loop
               Let
                  ⌜boundVarNames⌟ = [e]
               In
                  If ⌜guardCondIA⌟ Then
                     {[r] ← SOME ⌜patBrBodyIA⌟;  Break;}
                  End If
               End
            End Loop

            Return [r]
         End
       ⟩
      )
end;
```

ℙ : patAST → (IACTION,semMty) → env → (mty,IACTION,Name list,env)

(wild-card)
ℙ ⟦_⟧ (sIA,sMty) ρ = (sMty, ⟨[()]⟩, (), ∅)

(mutable binding)
ℙ ⟦**var** x⟧ (sIA,sMty) ρ =

```
    if mutable(sMty)
       then (sMty, ⟨[(sIA)]⟩, (x),  ∅[x ↦ sMty])
       else errorMsg("Pattern Matching: aliasing conflict")
```

(constant binding)

```
ℙ ⟦x⟧ (sIA,sMty) ρ = (sMty, ⟨[(sIA)]⟩, (x), ∅[x ↦ CON(sMty)])
```

(nullary value constructor)

```
ℙ ⟦C⟧ (sIA,sMty) ρ  =
  let
     Cmty = lookup(C,ρ)
  in
     if coercible(Cmty,sMty)
     then (Cmty, ⟨If C = sIA Then [()] Else []⟩, (), [])
     else errorMsg("Pattern Matching: type mismatch between
                    deconstructor and subject")
  end
```

(value construction)

```
ℙ ⟦C(p)⟧ (sIA,sMty) ρ  =
  let
     (~C_mut, FunType(~C_domMty,~C_coDomMty)) = lookup(~C,ρ)
     (pMty,valuesIA,boundVarNames,ρ′) = ℙ ⟦p⟧ [x] ρ[ [x] ↦ ~C_coDomMty ]
  in
     if coercible(~C_coDomMty,sMty)
     then
        (
           ~C_domMty,
           ⟨
              Case ~C(⌜sIA⌟) Of
                 SOME [xs] =>
                    Foldl ++ (Map (fn [x] => ⌜valuesIA⌟) [xs]) [] |
                 NONE => []
           ⟩,
           boundVarNames,
           ρ′
        )
     else errorMsg("Pattern Matching: type mismatch between
                    deconstructor and subject")
  end
```

(tuple)

```
ℙ ⟦(p1, ..., pn)⟧ (sIA,sMty) ρ  =
  if coercible(sMty, (CON Tuple_n_Type(cMty1, ..., cMtyn ))) then
     let
```

```
      (p1Mty, valuesIA1, boundVarNames1, ρ1)  =  ℙ ⟦p1⟧ [s1] ρ[ [s1] ↦ cMty1]
      ...
      (pnMty, valuesIAn, boundVarNames_n, ρn)  =  ℙ ⟦pn⟧ [sn] ρ[ [sn] ↦ cMtyn]
   in
      (
         Tuple_n_Type(p1Mty, ..., pnMty),
         ⟨
            Let
               [s1] = π1⌜(sIA)⌟
               ...
               [sn] = πn⌜(sIA)⌟
            In
               ⌜valuesIA1⌟ +++ ... +++ ⌜valuesIAn⌟
            End
         ⟩,
         boundVarNames1 ⧺̃ ... ⧺̃ boundVarNames_n,
         ρ1 ⧺̃ ... ⧺̃ ρn
      )
   end;
else errorMsg("Pattern Matching: type mismatch against
               tuple pattern")
```

(layered pattern)

```
ℙ ⟦x as p⟧ (sIA,sMty) ρ =
  let
     (pMty, valuesIA, boundVarNames, ρ') = ℙ ⟦p⟧ (sIA,sMty) ρ
  in
     (
       pMty,
       ⟨
          Map (Fn t => ((⌜sIA⌟)1 ++ t)) ⌜valuesIA⌟
       ⟩,
       (x)1 ⧺ boundVarNames,
       ρ'[x ↦ CON(sMty)]
     )
  end
```

Griffin allows user-defined deconstructors. As a consequence, pattern matching may have side-effects, thus the pattern-matching actions belong to the imperative action domain instead of the environment-enriching declaration action domain. The pattern matching is conducted in a way that the subject is matched against each pattern branch until a certain pattern branch is satisfied; otherwise the exception

"Pattern Matching Failure" is raised. When writing the semantic equations for pattern matching, special care should be taken to avoid causing the possible side effects more than once.

```
var x: int := 1;
fun inc y => y+1;

fun foo z =>
   case z of
      1 => output("1");  ||
      2 => output("2");
      _ => output("number greater than 2");

foo(inc(x));      ----  output string "number greater than 2"
                  ----  from an incorrect translation
```

A naive translation will increment the value of x to 4 and results in the string `number greater than 2` to be printed.

7.12 Generators, iterators, and comprehension expressions

In order to reduce the cost of programming, some high-level programming languages offer composite objects, such as sets, bags, maps, lists, and tuples. Powerful mathematical operations are available to manipulate these objects.

For example, the Griffin expression

```
[for i in 1..10 => i**2]
```

yields a list of the squares of integers from 1 to 10, in that order. "`1..10`" is a *generator*, "`i` **in** `1..10`" is an *iterator*, "**for** `i` **in** `1..10 => i**2`" is a *for-loop generator*, and the entire expression is a *list comprehension expression.*

An iterator is of the form "P **in** G", in which P is a pattern and G is a generator. An iterator loops through the elements yielded by the generator G, forces the pattern-matching between P and these elements, and provides a handle for later processing.

A *generator*, which can take on various syntactic forms, is a stream of values. Among these syntactic forms are the range "1..10" above, or an identifier which is of some aggregate type and a generator is defined for that aggregate type. For instance, if `S` is a set used in a context which requires a generator, an implicit coercion from a set to a generator is performed. The expression `e` **in** `S` is an iterator yielding the elements of `S` one at a time. To be exact,

```
for e in S
```

is really a shorthand for

```
for e in gen(S)
```

where `gen(S)` is a generator. The overloaded function `gen` is a coercion function which converts objects of an aggregate type to a stream of values of the aggregate component type. The instance of `gen` that takes a generator is the identity function:

```
fun gen(g: gen['a]) => g;
```

Definitions of `gen` for common aggregate types such as sets, bags, lists, tuples are provided in the standard language library.

Generators can be used in several contexts. For example, the `for loop` syntactic construct covered before can be extended to a generator. A Griffin `for-loop` generator has the syntax

$$\textbf{for}\ P_1\ \textbf{in}\ G_1,\ \ldots,\ P_n\ \textbf{in}\ G_n\ \texttt{|}\ C\ \texttt{=>}\ E$$

where P_i **in** G_i are iterators $\forall i \in 1 \ldots n$, C is a boolean-valued expression used as a guard, and E is an expression that yields the elements of the result. It is semantically equivalent to the following form:

$$\begin{array}{l} \textbf{for}\ P_1\ \textbf{in}\ G_1 \\ \quad \vdots \\ \qquad \textbf{for}\ P_n\ \textbf{in}\ G_n\ \texttt{|}\ C\ \texttt{=>}\ E \end{array}$$

Informally, the above generator matches pattern P_1 against the values generated by the corresponding generator G_1; for each successful match, its bindings will be added to the environment, and this process is repeated for pattern P_2 and G_2, and so on. Every time this process reaches the guard, i.e., every time a binding resulting from all successful matches is found, the guard is tested to decide whether we need to evaluate expression E and accumulate its result. The `for-loop` generator yields the sequence of values thus obtained. The formal action semantics description for `for-loop` generators is given at the end of this section.

Comprehension expressions allow us to define, construct, compare, and in general manipulate aggregates of values. They are mathematical constructs which make it relatively easy to represent complex control flow in a concise fashion. A comprehension expression is a high-level abstract control mechanism that hoists the looping capability from the conventional statement level (as it appears in most

imperative languages) to the expression level. One can place a generator inside square brackets to form a list comprehension expression as in

```
[for 1..10]
```

which is a list that contains integers from 1 to 10, in that order. To create a set of integers containing 1 through 10, we write

```
{for 1..10}
```

instead. Note that `[for 1..10]` is completely different from `[1..10]`, which is a list of length one whose sole element is a generator, namely, `1..10`.

A function in C or ML that takes an argument which is a set of sets, and returns the number of element sets of length two in that set would probably take tens of lines of code. In Griffin, the programming effort is significantly reduced. If `S` is a set of sets and `#` is the cardinality function on sets, the number of element sets of length two in `S` can be computed by the following expression,

```
#{for x as {_,_} in S => x}
```

or

```
#{for x in S | #x = 2 => x}
```

The syntax of a Griffin `for-loop` generator is

```
FLG  ::=  for iterators "|" C "=>" E
```

`iterators` denotes a sequence of iterators. To better present the semantic equations, we can rearrange the above grammar as

```
FLG  ::=  for B


B  ::=  "|" C "=>" E
     |   P "in" G "," B
```

in which the nonterminals FLG and B are for the `for-loop` generator and its body, respectively.

Recall that $\mathbb{P}$ is the semantic function for patterns defined in Section 7.11. It takes three parameters: pattern AST, subject, environment, and returns a triple consisting of an action yielding a list of tuple of values, list of binding names, and a new environment. The semantic equations for `for-loop` generator are:

```
𝔹 : GenBodyAST → env → (mty,IACTION)

𝔹 ⟦"|" C "=>" E⟧ ρ =
  let
     (Cmty, guardCondIA) = 𝔼 ⟦C⟧ ρ
     (bodyExpMty, bodyExpIA) = 𝔼 ⟦E⟧ ρ
  in
     (
       bodyExpMty,
       if coercible(Cmty,bool) then
          ⟨If ⌜guardCondIA⌟ Then [⌜bodyExpIA⌟] Else []⟩
       else
          ⟨Raise CondTypeNotBoolean⟩
     )
  end


𝔹 ⟦P in G "," B⟧ ρ =
  let
     ((genMut, genType), genIA) = 𝔼 ⟦G⟧ ρ
     (valuesIA,boundVarNames,ρ') =
        ℙ ⟦P⟧ (element,componentType(genType)) ρ
     (bodyExpMty,bodyIA) = 𝔹 ⟦B⟧ ρ'
  in
     (
       bodyExpMty,
       ⟨
```

```
        Let
            (stream,retrieve) = ⌜genIA⌟
            results = []
        In
            While True Loop
                Case retrieve(stream) Of
                    NONE => break;
                    SOME (element,newStream) =>
                        stream := newStream;

                        Let
                            x = ⌜valuesIA⌟
                        In
                            For e In x Loop
                                Let
                                    ⌜boundVarNames⌟ = e
                                In
                                    Let
                                        result = ⌜bodyIA⌟
                                    In
                                        results := results ++ result;
                                    End
                                End
                            End Loop
                        End
                End Case
            End Loop
            Return results
        End
    ⟩     )
end
```

7.13 Interlanguage conventions

A translation scheme should detail the interlanguage translation conventions used; a documentation of it is of considerable value in enhancing the readability of the target program. For example, the naming idiosyncrasy in source or target language like the combination of automatic declaration of variables and "I through N Rule" in Fortran should also be dealt with in the scheme. Also, avoidance of identifier names in the standard language library is crucial in practical applications [31].

Identifiers are normally considered to be strictly syntactic entities, there is no obvious conceptual difference between the syntactic construct of an identifier and its semantic counterpart. In language translation, however, it is quite reasonable to maintain such a distinction for identifiers [52]. The semantic counterpart of an identifier is thus referred to as a *semantic identifier*. By first mapping syntactic identifiers to semantic identifiers, the elaboration of identifier names in the target program is taken care in the microsemantics specification.

In G2A, syntactic identifiers appear in a Griffin program are translated into their corresponding semantic identifiers in TPOT. Suitable names in Ada will be chosen for them according to the name translation scheme in the microsemantics. Ada is case-insensitive, that means, it considers upper and lower case forms of a letter to be identical.

Lambda expressions have their arguments replaced by unique names, a process known as alpha-conversion [83].

7.14 From TPOTs to Ada ASTs : term-rewriting

The intermediate representation of G2A, TPOT, is transformed into the final Ada program in three phases. The first phase translates the TPOT into the extended Ada AST, $AdaAST^{+}$, which is an approximation of the TPOT but may not conform to the Ada syntax rules completely. $AdaAST^{+}$ allows some syntactic constructs which are not permitted in Ada, for example, assignment expressions.

The next phase a term rewriting system turns the $AdaAST^{+}$ into an syntactically correct Ada abstract syntax tree representation, $AdaAST$.

A term rewriting system (Σ, $\mathcal{R}$) consists of signature Σ and a set of (rewrite)

rules $\mathcal{R}$. The rules have the form $t_1 \rightarrow t_2$, where t_1 and t_2 are terms over Σ. Moreover, we must have:

- t_1 is not just a variable
- Every variable that occurs in t_2, must already occur in t_1. A rewrite rule may not introduce any new variables.

We impose appropriate restrictions to make the term rewriting system possess the *s*trong normalization property; that is, there is no infinite sequence of reductions

$$t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots$$

where $\rightarrow$ denotes one step reduction.

The proof of the termination of our system can be approached by defining termination functions that returns the outermost function symbol of a term, with symbols ordered by some precedence (a "precedence" is a well-founded partial ordering of symbols).

7.15 From Ada AST to textual representation

The last step of code generation is to produce the textual representation from an Ada AST. "UN-parsing" is the reverse process of parsing, in which an AST is the input and the textual representation of the AST is the output. UN-parsing of operator expressions is an interesting topic of its own and will be addressed in more detail in Section 7.15.1.

All code generation routines for pretty-printing are highly modularized functionals; they use state transitions to carry the configuration from one state to

another. The lowest-level pretty-printing routines takes cares of tabbing, indenting, unindenting, whitespaces, new line, etc. The next level routines take the low-level routines as parameters to provide higher-level functionality; at this level of abstraction, modifications to configurations are disallowed. One level up is the language-dependent level containing the Ada specific pretty-printing routines to generate Ada code for statements, expressions, declarations and other Ada syntactic constructs. The main routine traverses the rewritten *AdaAST*, at each node it invokes corresponding Ada printing routine to generate the Ada code.

7.15.1 UN-parsing operator expressions

Both precedence and associativity need to be taken into account in the translation of expressions involving operators. In the interest of readability, we would like to keep the generated code as *parenthesis-free* as possible.

Naively printed, expressions can be cluttered by lots of unnecessary parentheses, which makes it hard for human readers to comprehend. To increase readability of programs, we need to minimize the number of generated parentheses as long as the semantics is preserved, i.e., the resulting textual representation parsed according to the precedence and associativity of the operators should yield the same abstract syntax tree.

For example, assuming the usual precedence of arithmetic operators, the abstract syntax tree in Figure 7.20(a) and Figure 7.20(b) can be UN-parsed into `(1+2)*3` and `1+2*3` respectively. In the first case, subexpression `1+2` is enclosed in parentheses, because its parent is a higher-precedence operator "$*$"; in the second case, subexpression `2*3` does not need to be enclosed in parentheses, because

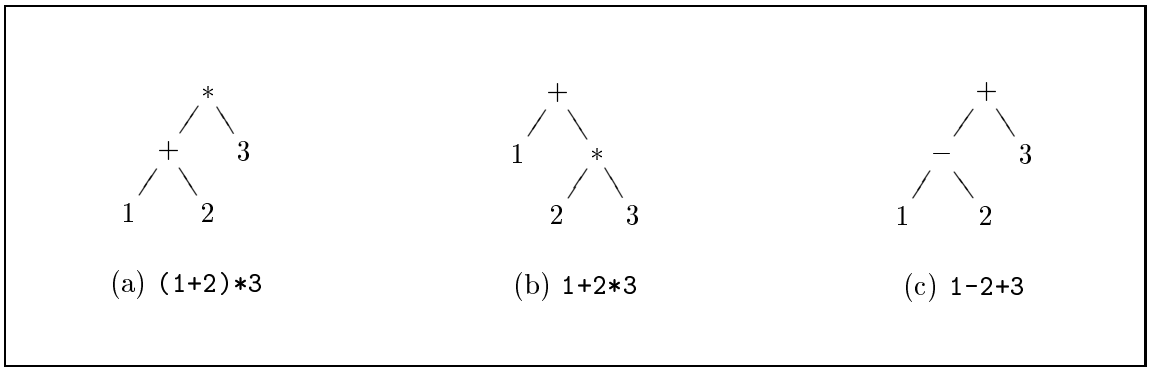


Figure 7.20: Example abstract syntax trees for operator expressions

its parent is a lower-precedence operator "+". The abstract syntax tree in Figure 7.20(c) is UN-parsed as `1-2+3`, because operator "−" and operator "+" have the same precedence and they are both *left* associative, and the subexpression `1-2` appears as the *left* subexpression of its parent. All the operators belonging to the same precedence group must have the same associativity, otherwise the parser will not be able to parse unambiguously expressions such as `1-2+3`. Associativity is a special syntactic issue that only exists among binary operators.

The above cases can be generalized to an algorithm for UN-parsing expressions into their textual representations, which generates only the *necessary* parentheses.

Let "⇒" denote the function application operator. If we define:

$$\boldsymbol{U} = \text{ set of all unary operators}$$
$$\boldsymbol{B} = \text{ set of all binary operators}$$
$$\boldsymbol{Op} = \boldsymbol{U} \cup \boldsymbol{B}$$
$$\mathbb{N} = \{1, 2, \ldots, n, n+1, n+2, n+3\}$$

The following $\mathcal{P}$ function whose type is $\boldsymbol{Op} \rightarrow \mathbb{N}$, assigns integer precedence to

operators:

$$\begin{aligned} \mathcal{P}[\![\Rightarrow]\!] &= n+2 \\ \mathcal{P}[\![uop]\!] &= n+1 \qquad \text{if} \quad uop \in \boldsymbol{U} \\ \mathcal{P}[\![bop]\!] &\in \{1,\ldots,n\} \qquad \text{if} \quad bop \in \boldsymbol{B} \end{aligned}$$

We illustrate the algorithm with the following grammar of expressions:

$$e \;:=\; uop\ e \mid e_l\ bop\ e_r \mid aexp \mid e_0(e_1, e_2, ..., e_n)$$

where $uop \in \boldsymbol{U}$, $bop \in \boldsymbol{B}$, $aexp$ is an atomic expression such as a variable or a constant literal, and $e_0(e_1, e_2, ..., e_n)$ is the form of function application.

Function applications have the highest precedence. All unary operators belong to the same next highest precedence level, and all binary operators have lower precedence.

We also extend the precedence function to expression parameters in a natural way: it maps expressions to the precedences of their top-level operators, if they have one; otherwise their precedence is $n+3$ (for the case of atomic expressions). In addition, each binary operator bop also has an associativity $\mathsf{assoc}(bop) \in \{\textsc{Left}, \textsc{Right}, \textsc{Non}\}$.

Based on the precedence and associativity of operators, our UN-parsing algorithm can be formalized as a function $\mathcal{U}$ that computes the textual representation of an expression (Figure 7.21). The auxiliary function paren parenthesizes a string, e.g., paren("1+2") = "(1+2)", and $+\!\!+$ is the overloaded concatenation function defined on the disjoint sum of strings, atomic expressions, operators, and returns a string.

To implement the above algorithm, a table (possibly user-configurable) is required by the code generation routine. The table documents the precedence level,

$$
\begin{array}{rcl}
\mathcal{U}[\![uop\ e]\!] & = & uop\ +\!\!+ (\textbf{if}\ \mathcal{P}[\![e]\!] \geq n+1\ \textbf{then}\ \mathcal{U}[\![e]\!]\ \textbf{else}\ \mathsf{paren}(\mathcal{U}[\![e]\!])) \\
\mathcal{U}[\![e_l\ bop\ e_r]\!] & = & (\textbf{if}\ (\mathcal{P}[\![e_l]\!] < \mathcal{P}[\![bop]\!])\ \textbf{or} \\
 & & \quad (\mathcal{P}[\![e_l]\!] = \mathcal{P}[\![bop]\!]\ \textbf{and}\ \mathsf{assoc}(bop) \neq \textsc{Left}) \\
 & & \textbf{then}\ \mathsf{paren}(\mathcal{U}[\![e_l]\!]) \\
 & & \textbf{else}\ \mathcal{U}[\![e_l]\!]) \\
 & & +\!\!+\ bop\ +\!\!+ \\
 & & (\textbf{if}\ (\mathcal{P}[\![e_r]\!] < \mathcal{P}[\![bop]\!])\ \textbf{or} \\
 & & \quad (\mathcal{P}[\![e_r]\!] = \mathcal{P}[\![bop]\!]\ \textbf{and}\ \mathsf{assoc}(bop) \neq \textsc{Right}) \\
 & & \textbf{then}\ \mathsf{paren}(\mathcal{U}[\![e_r]\!]) \\
 & & \textbf{else}\ \mathcal{U}[\![e_r]\!]) \\
\mathcal{U}[\![aexp]\!] & = & aexp \\
\mathcal{U}[\![e_0(e_1,\ldots,e_n)]\!] & = & \textbf{if}\ \mathcal{P}(e_0) \geq n+2\ \textbf{then}\ \mathcal{U}[\![e_0]\!]\ \textbf{else}\ \mathsf{paren}(\mathcal{U}[\![e_0]\!]) \\
 & & +\!\!+\ \text{“(”}\ +\!\!+\mathcal{U}[\![e_1]\!]\ +\!\!+\ \ldots\ +\!\!+\mathcal{U}[\![e_n]\!]\ +\!\!+\ \text{“)”}
\end{array}
$$

Figure 7.21: The UN-parsing algorithm that minimizes number of generated parentheses

and associativity of operators in the target language.

As a consequence of the above algorithm, we will not be able to keep the similarity between the source and target program if there are unnecessary parentheses in the source program. Since the generated code is more readable without the unnecessary parentheses, we consider our approach preferable. If we really want to keep the similarity, we can annotate each node with an extra boolean field. If the value of the field is true, surrounding parentheses will always be generated regardless of the precedence or associativity of the node.

7.15.2 Monadic-style multi-level pretty-printer

Software reuse is plainly visible in functional programs. Most functions in functional programs capture very general programming idioms that are useful in almost any context. But it is just as important to define and use application specific idioms. The functional programmers should approach a new application by seeking to identify the programming idioms common in that application area, and to define them as functions (probably higher order). Each particular application program should then be built by so far as possible combining these functions, rather than writing "new code" (for this reason, such functions are often called combinators). The benefits of such an approach are very rapid programming, once the library of idioms is defined, and very often that application programs are correct the first time, since they are built by assembling correct components.

Almost every program which manipulates symbolic data needs to display the data to the user at some point — whether it is for internal compiler debugging, or a program transformer writing its output. The problem of displaying symbolic, and especially tree structured data, is a recursive one. A pretty printer's job is to lay out structured data appropriately. Pretty-printing is complicated because the layout of a tree node cannot just be inferred from its form. Instead, a pretty-printer must keep track of much contextual information.

Our pretty printer is language independent at the lower levels; only the highest level deals with language-specific syntactic constructs. It is a variant of the pretty-printers originally designed by John Hughes [45] used in both the Chalmers and Glasgow Haskell compilers. To give a flavor of the monadic-style multi-level pretty printer, some interesting code snippets are shown in Figure 7.22 (signature) and

Figure 7.23 (implementation).

The pretty printer is stratified in three layers. At the lowest level (*pretty printer core*), the representation of *state* (or *configuration*) and various pretty printers are defined. Essentially a pretty printers is a state transformer (of abstract type `state -> state`). Functions that change the state belong to this level. Of particular importance is the concatenation operator `++`, which is defined as the function composition of the state transformers, thus is *mathematically associative*.

Functions defined at the lowest level have access to the representation of the state. Such details are abstracted away to the higher-level pretty printing routines. Only a few primitives are provided at this level.

Built upon the lowest level primitive pretty printer and combinators are higher-level pretty printing routines at the second level such as `seqMap`, which pretty-prints a sequence of elements. It is of type `pp * pp * pp -> ('a ->pp) -> 'a list -> pp`. The first argument is a triple `(prologue,separator,epilogue)`. `prologue` and `epilogue` specify what should be pretty-printed before and after the sequence, while `separator` specify what should be pretty-printed between elements. The second argument is the pretty-printer for elements in the third arguments. For example, to pretty-print a list of integers enclosed in square brackets, one can write `seqMap (lb,noOp,rb) intPP listOfInt`, and sequence of statements can be pretty-printed with `seqMap (noOp,semi,noOp) stmtPP listOfStmt`, where `intPP` and `stmtPP` are the pretty-printers for integer and statement, respectively.

At the highest level, language-specific pretty printers are defined.

```
(*   Pretty printer core   *)

     type state
     type pp

     val indentation : int

     val $   :  string  -> pp    (*  prefix a space to the string  *)
     val @@  :  string  -> pp    (*  identifier pretty printer  *)

     val tab       :  pp
     val indent    :  pp
     val unindent  :  pp
     val nl        :  pp
     val space     :  pp
     val prog      :  pp -> string
     ...


(*   Higher-level pretty printer   *)

     val ++  :  pp * pp -> pp    (*  for concatenation  *)

     val string  :  string -> pp
     val integer :  int    -> pp

     val comma  : pp  (* , *)
     val dquote : pp  (* `` *)

     val lp : pp  (* ( *)
     val rp : pp  (* ) *)

     val noOp   :  pp
     val keywd  :  string -> pp
     val block  :  pp -> pp      (*  first indent, then unindent  *)
     val seq    :  pp * pp * pp -> pp list -> pp
     val seqMap :  pp * pp * pp -> ('a -> pp) -> 'a list -> pp

     val paren   : pp -> pp
     val bracket : pp -> pp
     val brace   : pp -> pp
     ...


(*   Language-dependent level   *)

     val stmtPP : AdaAST.stmt -> pp
     ...
```

Figure 7.22: Signature of the monadic-style multi-level pretty-printer

```
(*   Pretty printer core   *)
type state = int * int * string
type pp = state -> state
val indentation = 3
fun $ s (i:int,p,t)  = (i, p+1+size s, t ^ " " ^ s)
fun @@ s (i:int,p,t) = (i, p+size s, t^s)  (* i:indentation, p:position *)
fun tab (s as (i,p,t)) = if p >= i then s else (i, p, t ^ whiteSpaces(i-p))
fun nl (i,p,t)       = (i, 0, t ^ "\n")
fun indent (i,p,t)   = (i+indentation,p,t)
fun unindent (i,p,t) = (i-indentation,p,t)
fun space (i,p,t)    = (i, p+1, t ^ (whiteSpaces 1))
fun prog (pp:pp)     = (#3(pp(0,0,"")))
...

(*   Higher-level pretty printer   *)
fun pp1 ++ pp2 = pp2 o pp1
val comma      =   @@ ","
val dquote     =   @@ "\""
val lp         =   @@ "("
val rp         =   @@ ")"
fun string  s  = dquote ++ (@@ s) ++ dquote
fun integer i  = @@ (Int.toString  i)
fun noOp    pp = pp

fun seq (left,middle,right) (list : pp list) =
   let
      fun printSeqElems []      = noOp
       |  printSeqElems [x]     = x
       |  printSeqElems (x::xs) = x ++ middle ++ printSeqElems xs
   in  left ++ (printSeqElems list) ++ right  end

fun seqMap (left,middle,right) f xs = seq (left,middle,right) (map f xs)

fun block   pp = indent ++ pp ++ unindent
fun line    pp = tab ++ pp ++ nl
fun keywd   kw = @@kw
fun paren   pp = lp ++ pp ++ rp
fun bracket pp = lb ++ pp ++ rb
fun brace   pp = lbr ++ pp ++ rbr
...

(*   Language-dependent level   *)
fun stmtPP (Block blockIdOpt,decls,stmts,exnHandlers) =
    (blockIdOptPP blockIdOpt)                                  ++
    (blockDeclsPP decls)                                       ++
    (kwLine KWbegin)                                           ++
    (block (seqMap (noOp,noOp,noOp) stmtPP stmts))             ++
    (exnHandlersPP exnHandlers)                                ++
    (line ((@@KWend) ++ (endLabelPP blockIdOpt) ++ semi))
...
```

Figure 7.23: Implementation of the monadic-style multi-level pretty-printer

Chapter 8

Conclusion and future work

The major purpose of this dissertation research is to propose a new approach to the process of designing, developing, using, maintaining, and documenting prototyping languages. We advocate that employing a translator from a prototyping language to another high-level language serves as a fast and inexpensive testbed for the entire process.

We have demonstrated a model for language translation based on action semantics for quick development of language translators. A strength of our model is that a semantic description of the source language can be given formally in a concise manner inside the translator itself. The layered and modular structure of this translator (G2A) makes it more abstract, more readable than existing translators that we know of, and readily accommodates changes to allow the translator to evolve with the language. The type system used in the static analyzer of G2A is parameterizable, which is a useful feature for experimenting with various type systems, especially in the development stages of a prototyping language. This better description of the semantics of a language makes it easier for implementors to pro-

vide a correct implementation, and for users or theoreticians to gain insights into the language. Furthermore, this model facilitates mathematical reasoning and a correctness proof of the entire translation process. Our implementation experience indicates that the cost of describing the semantics of a language in action notation is modest, certainly much lower than the price paid for unsound features and misunderstanding that can result from informal descriptions. We hope that our work will help to convince designers of prototyping languages to regard an action-semantics based language description as a reasonable support for the development efforts, like other essential tools of the language, rather than as an option too expensive to consider. Although we have only implemented a translator from Griffin to Ada95, the generality of the system should apply to other language translators as well.

We have acquired a collection of techniques for the translation of certain non-trivial high-level features of prototyping languages and declarative languages into efficient procedural constructs in imperative languages like Ada95, while using the abstraction mechanism of the target languages to maximize the readability of the target programs. In particular, we translate Griffin existential types into Ada95 using its object-oriented features, based on coercion calculus. This translation is actually more general, in that one can add existential types to a language (with slight change of its syntax) supporting object-oriented paradigm without augmenting its type system, through intra-language transformation. We also present a type-preserving translation of closures which allows us to drop the whole-program-transformation requirement.

Most of the chapters include sections giving a few of the ideas we have not been able to follow up on yet. Overall, there are some possible extensions of this work

which will be described in the following sections.

8.1 Dropping the whole-program-transformation requirement

A significant restriction of our system is that it needs all the Griffin program components available before the translation process starts. Our translations of several language features require access to the entire program before the translation starts, such as the constraints collecting (Section 5.1.4), existential type translation (Section 7.2), and structural type equivalence in Griffin versus name equivalence in Ada95 (Section 6.2).

8.2 Separate compilation

Modularization, the division of a program into a number of relatively manageable modules, is an invaluable mechanism of programming languages to support "programming in the large".

At the present time, Griffin lacks a module system, therefore separate compilation is not taken into consideration. With a modicum of extra apparatus, G2A can embrace modular design. All the symbols used in a module have to be explicitly introduced in one of the following ways: in the module itself, in an external specification, or in a reference module (possibly along with some language-specific conventions).

8.3 Action semantics based type system

To obtain a coherent documentation style for the entire system, we can also express the type system itself in action semantics. In this way, we also "stage" an action semantics definition of a programming language into static and dynamic stages as most other formal description methods do.

In our implementation, the Griffin type system [70] is an external module. Auxiliary functions are established to invoke various functions for type inference or type checking purposes.

To obtain an even stronger flavor of action semantics in our implementation, we can develop a type system for actions based on *types* and *kinds*. The types within a kind can be partially ordered to reflect subtyping. Such a type system description would be similar to the one presented in [29, 30], wherein a type system and its interpretation support an ML-style type inference algorithm for action expressions. The authors describe a unification-based, decidable type inference algorithm for action semantics in great detail.

8.4 Extracting front-end from semantic specifications

It is possible and indeed preferable to derive lexers and parsers automatically from semantic specifications. Using the system ASF+SDF [89] developed at Inria, one specifies concrete syntax rather than abstract syntax in the semantic equations, and a lexer and a parser can be generated from the semantic specification accordingly. The automatically derived front-end could be further used in the development of other tools such as the language debugger.

Another possibility is the approach used in [52] in which the abstract syntax (part of the semantic specification) is specified, but with some implicit naming rules and auxiliary documentation, the front-end is derivable from the semantic specification.

Another minor point is that we can also make the operators of the source or target language more dynamic by providing a mapping documenting their precedence and associativity. The mapping would be read by the front-end generator to produce the eventual lexer and parser.

8.5 Further development of the theory of action notation

Action equivalence provides sufficient abstraction to verify various simple algebraic laws. However, the general theory of action notation has not yet been fully developed, and at the time of writing it is not known whether the degree of abstraction provided is high enough for general use in program verification [64].

8.6 First-class polymorphism

Another interesting extension is to take into consideration of first-class polymorphism [48]. In most strongly typed languages, higher-order polymorphic functions are not implemented because, in general, type inference, or even type checking, is undecidable. However, Griffin imposes certain syntactic restrictions to support first-class polymorphism, which ensures the decidability of the type system.

Appendix A

Macrosemantic specification of Griffin for-loop generators

A.1 Abstract syntax trees of Griffin for-loop generators

```
functor AstFunc (structure Lit  : LIT
                 structure Type : TYPE
                 structure Prim : IR_PRIM
                 sharing Lit.Print = Type.Print) : AST =
struct
   ⋮
   datatype expAST =
      ⋮
      ForGenAST of generatorAST
      ⋮
   and ...
      ⋮
   withtype iteratorAST =
      {
         pat  : patAST,
         expr : exprAST
      }
   and generatorAST =
      {
         iterators  :  iteratorAST list,
```

```
        cond       :  bool
      }
   ⋮
end
```

A.2 Grammar rules of Griffin for-loop generators

```
%nonterm
   ⋮
   |  expr              of  expAST
   |  generator_expr    of  generatorAST
   |  iterator          of  iteratorAST
   |  iterators         of  iteratorAST list
   ⋮

%%
(* ----------- *)
(*    rules    *)
(* ----------- *)

expr :
   ⋮
   FOR  generator_expr
      (ForGenAST generator_expr)    |
   ⋮

generator_expr :
   iterators    |
      ({iterators = iterators, cond = true})
   iterators  EqGt  expr
      ({iterators = iterators, cond = expr})

iterators :
   iterator
      ([iterator])                |
   iterator  Comma  iterators
      (iterator :: iterators)

iterator :
   pat  IN  expr
      ({pat = pat, expr = expr})    |
   pat  IN  expr  Vbar  expr
      ({pat = pat, expr = expr})
```

A.3 Semantic equations of Griffin for-loop generators

The semantic equations for Griffin `for-loop` generator are:

```
B : GenBodyAST -> env -> (mty,IACTION)

B  [[ "|"  C  "=>"  E ]]  env =
   let
      (Cmty, guardCondIA) = E [[C]] env
      (bodyExpMty, bodyExpIA) = E [[E]] env
   in
      (
        bodyExpMty,
        if coercible(Cmty,bool) then
           IfThenElse(guardCondIA, [bodyExpIA], [])
        else
           Raise(CondTypeNotBoolean)
      )
   end


B  [[ P  "in"  G  ","  B ]]  env  =
   let
      element  = genSym()
      stream   = genSym()
      retrieve = genSym()
      results  = genSym()
      result   = genSym()
      x        = genSym()
      e        = genSym()

      ((genMut, genType), genIA) = E [[G]] env
      (valuesIA,boundVarNames,env') =
         P [[P]] (element,componentType(genType)) env
      (bodyMty,bodyIA) = B [[B]] env'
   in
      (
         bodyExpMty,
         LetInEnd
         (
          [
           Bind((stream,retrieve),genIA),
           Bind(results,[])
          ],
          [
           WhileLoop(
            True,
            [
             Case (
               FuncCall(retrieve,stream),
```

```
            [
             (NONE, [Break]),
             (
               SOME (element,newStream),
               [
                Assign(stream,newStream),
                LetInEnd (
                 [Bind(x,valuesIA)],
                 [
                  ForLoop(
                   e,
                   x,
                   LetInEnd(
                    [boundVarNames = e],
                    [
                     LetInEnd(
                      [Bind(result,bodyIA)],
                      [Assign(results, results ++ result)])
                    ]))])])])]),
          Return(results)
         ]
       )
    )
end
```

Bibliography

[1] Ada 95 rationale, September 1995.

[2] Ada 95 reference manual, 1995.

[3] Niki Afshartous and Malcolm C. Harrison. Expressing concurrency in Griffin. In *IEEE International Conference on Parallel and Distributed Systems*, 1996.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[5] Andrew W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[6] J. C. M. Baeten and W. P. Weijland. *Process algebra.* Cambridge University Press, 1990.

[7] Ted Baker and Offer Pazy. Real-time features for Ada 9X. In *IEEE Real-time system symposium*, pages 172–180, 1991.

[8] H. P. Barendregt. *The Lambda Calculus.* North Holland, 1984.

[9] J. G. P. Barnes. *Programming in Ada95.* International Computer Science Series, 2nd edition, 1998.

[10] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice Hall, New York, 1st edition, 1990.

[11] J. F. Bartlett. SCHEME → C: a portable Scheme-to-C compiler. Research Report 89/1, DEC Western Research Laboratory, Palo Alto, CA, 1989.

[12] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, 9–11 June 1997.

[13] Nick Benton, Andrew Kennedy, and George Russell. Compiling standard ML to Java bytecode. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 129–140, 27–29 September 1998.

[14] W Bischofberger and G. Pomberger. *Prototyping-Oriented Software Development - Concepts and Tools*. Springer, Berlin, 1992.

[15] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 197–206, Albuquerque, New Mexico, June 1993. ACM Press.

[16] V. Breazu-Tannen and R. Subrahnanyam. Logical and computational aspects of programming with sets/bags/lists. In Javier Leach Albert, Burkhard Monien, and Mario Artalejo Rodríguez, editors, *Proceedings of Automata, Languages and Programming (ICALP '91)*, volume 510 of *LNCS*, pages 60–75, Berlin, Germany, July 1991. Springer.

[17] Alexander Brodsky and Victor E. Segal. The C3 constraint object-oriented

database system: An overview. In Volker Gaede et al., editors, *Constraint Databases and Their Applications, Second International Workshop on Constraint Database Systems*, volume 1191 of *Lecture Notes in Computer Science*, pages 134–159, Delphi, Greece, 11–12 January 1997. Springer.

[18] R. Budde, K. Kautz, K. Kuhlenkamp, and H. Zuellighoven. *Prototyping; An Approach to Evolutionary System Development.* Springer, Berlin, 1992.

[19] J. Cai and R. Paige. Towards increased productivity of algorithm implementation. In *Proc. ACM SIGSOFT*, pages 71–78, Dec. 1993.

[20] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):480–521, December 1985.

[21] Luca Cardelli. Basic polymorphic typechecking. In *Science of Computer Programming*, April 1987.

[22] Connell and Shafer. *Structured Rapid Prototyping.* Prentice Hall, 1989.

[23] Regis Cridlig. An optimising ML to C compiler. In David MacQueen, editor, *ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, June 1992. ACM Press.

[24] J. Darlington and R. M. Burstall. A system which automatically improves programs. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 479–485, Standford, CA, August 1973. William Kaufmann.

[25] Nachum Dershowitz. A taste of rewriting. In *Lecture Notes in Computer Science 693*, pages 199–228. Springer-Verlag, 1993.

[26] Dominic Duggan and John Ophel. On type-checking multi-parameter type classes. 1995.

[27] Hartmut Ehrig and Bernd Mahr. *Fundamentals of algebraic specification 1: equations and initial semantics.* Springer-Verlag, Berlin, 1985.

[28] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, USA, 1990.

[29] Susan Even and David A. Schmidt. Category sorted algebra-based action semantics. *Theoretical Computer Science*, 77:73–96, 1990.

[30] Susan Even and David A. Schmidt. Type inference for action semantics. Technical report, Kansas State University, Manhattan, KS, 1990.

[31] S. I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schryer. A Fortran-to-C converter. Technical Report 149, AT&T Bell Laboratories, May 1992.

[32] Matthias Felleisen. On the expressive power of programming languages. In N. Jones, editor, *ESOP '90 3rd European Symposium on Programming, Copenhagen, Denmark*, volume 432 of *Lecture Notes in Computer Science*, pages 134–151. Springer-Verlag, New York, N.Y., May 1990.

[33] Anthony J. Field and Peter G. Harrison. *Functional programming.* Addison Wesley, 1988.

[34] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Prin-*

ciples of Programming Languages (POPL'95), pages 379–392, San Francisco, California, January 22–25, 1995. ACM Press.

[35] C. N. Fischer and Jr. LeBlanc, R.J. *Crafting a Compiler*. Benjamin/Cummings Series in Computer Science. Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.

[36] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[38] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, November 1996.

[39] Malcolm C. Harrison. The Griffin prototyping language. In *Proceedings of DARPA PI meeting*, 1991.

[40] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA), La Jolla, California*. ACM, ACM Press, June 1995.

[41] Martin Hofmann and Benjamin Pierce. Positive subtyping. Technical Report ECS-LFCS-94-303, Department of Computer Science, University of Edinburgh, Edinburgh, U.K., September 1994. An extended abstract will appear in the

POPL'95 proceedings. Available by anonymous ftp from `ftp.dcs.ed.ac.uk` in file `pub/bcp/pos.ps.Z`.

[42] Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. Parallelization in calculational forms. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, 19–21 January 1998.

[43] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict purely functional language. Technical Report YALEU/DCS/RR-777, Yale University, Department of Computer Science, March 1992.

[44] Gerard Huet and Bernard Lang. Proving and applying program transformation expressed with second-order patterns. In *Acta Informatica*, pages 31–55, 1978.

[45] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[46] L. S. Hunt. A Hope to FLIC translator with strict analysis. Master's thesis, Department of Computing, Imperial College, University of London, 1986.

[47] Mark P. Jones. ML typing, explicit polymorphism and qualified types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 56–75. Springer-Verlag, April 1994.

[48] Mark P. Jones. First-class polymorphism with type inference. In *Conference*

Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 483–496, Paris, France, 15–17 January 1997.

[49] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Berlin: Springer-Verlag, 1985.

[50] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice Hall, 1993.

[51] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1988.

[52] Peter Lee. *Realistic compiler generation.* The MIT Press, 1988.

[53] Peter Lee and Uwe Pleban. A realistic compiler generator based on high-level semantics. In *Conference record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 284–295, 1987.

[54] David MacQueen. Using dependent types to express modular structure. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 277–286. ACM, January 1986.

[55] Ernest Manes and Michael Arbib. *Algebraic Approaches to Program Semantics.* Springer, 1986.

[56] L.G.L.T. Meertens, editor. *Program specification and transformation. Proceedings of the IFIP TC2/WG 2.1 working conference on program specification and transformation*, April 1986.

[57] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system science*, 17(3):348–375, 1978.

[58] Robin Milner, Mads Tofte, and Robert Harper. *The definition of standard ML*. The MIT Press, 1990.

[59] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg Beach, Florida, 21–24 January 1996.

[60] J. C. Mitchell. Coercion and type inference. In *11th Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[61] John Mitchell. *Foundations for programming languages*. The MIT Press, 1996.

[62] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.

[63] Peter D. Mosses. A basic semantic algebra. In *Lecture Notes in Computer Science 173*, pages 87–107. Springer-Verlag, 1984.

[64] Peter D. Mosses. *Action semantics*. Cambridge University Press, 1992.

[65] Peter D. Mosses and David A. Watt. The use of action semantics. In *Proceedings of the IFIP TC2 Working Conference on Formal Descriptions of Programming Concepts III*, 1987.

[66] Peter D. Mosses and David A. Watt. Pascal action semantics, version 0.6, March 1993.

[67] H. R. Nielson and F. Nielson. Semantics directed compiling for functional languages. In *Proceedings of 1986 ACM Conference on LISP and Functional Programming*, pages 249–257, 1986.

[68] Hanne Rijs Nielson and Flenning Nielson. *Semantics with Applications : A Formal Introduction.* John Wiley & Sons, Chichester, 1992.

[69] John Ophel and Dominic Duggan. Multi-parameter parametric overloading. 1995.

[70] Edward Osinski. *Implementation of a Prototyping Language.* PhD thesis, Computer Science Department, New York University, New York, NY, June 1999.

[71] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[72] Lawrence Paulson. A semantics-directed compiler generator. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 224–239, 1982.

[73] Lawrence Paulson. *ML for the Working Programmer.* Cambridge University Press, second, paperback edition, 1992.

[74] Gordon Plotkin. Structural operational semantics. Lecture Notes, DAIMI FN-19, Aarhus University, Denmark, 1991.

[75] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design.* Prentice Hall, 1991.

[76] Wolfgang Schreiner. Compiling a Functional Language to Efficient SACLIB C. Technical Report 93-49, RISC-Linz, Johannes Kepler University, Linz, Austria, 1993.

[77] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: an Introduction to SETL.* Springer-Verlag, 1986.

[78] Ravi Sethi. *Programming Languages: Concepts and Constructs.* Addison Wesley, 1989.

[79] Kenneth Slonneger and Barry L. Kurtz. *Formal syntax and semantics of programming languages.* Addison Wesley, 1995.

[80] Marvin Solomon. *On type definitions with parameters.* PhD thesis, Computer Science Department, University of Wisconsin, Madison, WI, June 1977.

[81] I. Sommerville. *Software Engineering.* Addison Wesley, Wokingham, England, 4th edition, 1992.

[82] J. M. Spivey. *Understanding Z. A specification language and its formal semantics.* Cambridge University Press, 1988.

[83] Jr. Steele, G.L. Rabbit: A compiler for Scheme. Technical Report AITR-474, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, May 1978.

[84] Joseph E. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory.* The MIT Press, 1977.

[85] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[86] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

[87] R. D. Tennent. *Semantics of programming languages.* Prentice Hall, 1991.

[88] Andrew Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 1997.

[89] M. G. J. Van den Brand and E. Visser. The ASF+SDF meta-environment: Documentation tools for free! *Lecture Notes in Computer Science*, 915, 1995.

[90] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the IEEE symposium on Logic in Computer Science*, 1987.

[91] David A. Watt. Executable denotational semantics. *Software: Practice and Experience*, 16(1):13–43, 1986.

[92] David A. Watt. An action semantics of standard ML. In *Proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics*, 1988.

[93] Glynn Winskel. *The formal semantics of programming languages.* The MIT Press, 1993.

[94] Zhe Yang. Encoding types in ML-like languages. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 289–300, 27–29 September 1998.